Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Aprendizaje en el funcionamiento de la herramienta SDSoC de Xilinx para diseños basados en dispositivos SoC

Autor: Daniel Merino Pérez

Tutor/es: Ignacio Bravo Muñoz

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Aprendizaje en el funcionamiento de la herramienta SDSoC de Xilinx para diseños basados en dispositivos SoC

Autor: Daniel Merino Pérez

Tutor: Ignacio Bravo Muñoz

TRIBUNAL:

Presidente: Alfredo Gardel Vicente

Vocal 1º: Pedro Martín Sánchez

Vocal 2º: Ignacio Bravo Muñoz

FECHA: 24-septiembre-2019

Índice

Índice de figuras

Resumen

En este TFG, se expondrán los distintos pasos a seguir para crear proyectos que se puedan implementar en un SoC haciendo uso de la herramienta SDSoC. En ellos, se hará uso del particionado HW/SW para poder acelerar el algoritmo que se esté desarrollando.

Para comprobar el gran potencial de esta herramienta y poder ilustrar los distintos pasos que se deben seguir para poder llegar a un proyecto que se pueda implementar. Se usará como ejemplo el algoritmo PCA aplicado a un set de imágenes cargadas en memoria. Para ello, se usarán librerías de OpenCV y un Sistema Operativo.

Palabras clave: HLS, Particionado HW/SW, SDSoC, PCA, SoC.

Summary

This Final Project addresses the different steps to implement a whole approach under a System on Chip (SoC) device. The use of SDSoC tool provided by Xilinx allows an easy way to make a HW-SW partition in order to speed up the algorithm to be implemented.

To validate the use for the speed-up of a user case, the Principal Component Analysis (PCA) technique is implemented for a Zynq device with SDSoC. An approach based on OpenCV libraries and a Linux Operating System is done to validate the benefits of this tool.

Keywords: HLS, HW/SW partition, SDSoC, PCA, SoC.

Resumen Extendido

El TFG se inicia justificando el porqué es interesante aprender a usar está novedosa herramienta y poniendo en contexto la situación actual del uso de la herramienta SDSoC. La cual se va a explicar cómo usarla en este trabajo. Además de mostrar el gran potencial de esta herramienta.

Tras ello, se explicará tanto las principales características del programa SDSoC como de todos los programas, desarrollados también por Xilinx, de los que hace uso para poder tener su gran potencial. Estos programas son: Vivado y HLS. Vivado se usará para, que de forma automática y transparente al usuario de SDSoC, realizar las conexiones necesarias para un correcto funcionamiento del algoritmo entre memoria, la unidad de procesado (donde se ejecutará la parte SW del algoritmo) y la FPGA incluida en el SoC (donde se ejecutará la parte HW). HLS se emplea para la implementación en Hardware de las funciones en C++ que se quieran acelerar. Para que esta aceleración sea óptima, se empleará el uso de sentencias pragma para indicar al compilador cómo debe hacer la implementación. Se hará uso tanto de las propias de HLS como de SDSoC.

Una vez expuesto el Software, se presentarán las especificaciones del HW empleado para este trabajo. Concretamente se usa la tarjeta de desarrollo ZedBoard. La cual tiene como núcleo central el SoC Zynq 7000 de Xilinx.

A continuación, se encuentra una sección que hará las veces de tutorial para poder familiarizarse con la herramienta SDSoC. Además de dar unos primeros pasos y unas indicaciones de las funcionalidades más usadas en el desarrollo de aplicaciones al usar este programa, se indica cómo incluir un sistema operativo corriendo en el SoC, en este caso Petalinux, y librerías de visión artificial. Tanto las OpenCV estándar como las desarrolladas por Xilinx, diseñadas para optimizar su implementación en HW. Estas últimas no se han implementado en la solución final porque no aportaban una mejora apreciable. Aunque se hace una explicación de ellas para poder comprender mejor la forma de trabajar con ellas y como incluirlas en el proyecto por si a la hora de construir otro algoritmo si resultan interesantes su uso.

Tras haber explicado tanto los programas como el HW a usar, se pasa a realizar un ejemplo práctico para poder mostrar todo el potencial del programa. El algoritmo que se ha decidido usar en este trabajo ha sido el PCA (*Principal Components Analysis*). Primero se hace una pequeña explicación de que es un algoritmo que se va a usar para detectar objetos nuevos en la imagen que se está procesando en comparación a un fondo definido al inicio de la ejecución del algoritmo. Para ello, se hará uso de la reducción de la dimensionalidad pasando por un espacio transformado con una dimensionalidad justificada en ese apartado.

A este algoritmo, se le aplicará una herramienta de perfilado para ver cuáles son las funciones candidatas a ser aceleradas mediante HW. Se ha decidido que estas sean las que más tiempo de CPU consuman. Una vez seleccionadas se someterán a un proceso de optimización HW que se detallará para cada una de las funciones. Además se hace una justificación de los tipos de datos elegidos para cada una de las variables usadas en el programa y de la elección de usar librerías de OpenCV y un SO.

Para poder observar el potencial de esta herramienta y de lo que nos puede brindar, se presentan y justifican unos datos de ejecución del programa sin y con particionado. De ellos se extrae la aceleración obtenida para dos casos: sin y con actualización de fondo. En este punto se puede ver que el programa, al construir el proyecto, construye todos los mecanismos de sincronización necesarios para que pueda haber ejecución HW y SW en paralelo y así conseguir mayor aceleración del algoritmo.

Por último, se expondrán una serie de conclusiones extraídas de la ejecución del proyecto y se comentarán una serie de trabajos futuros para poder eliminar la mayor sobrecarga del sistema y hacer que esta aceleración sea lo mayor posible y el algoritmo se ejecute lo más rápido posible. También se propondrán probar la eficacia del algoritmo con otros set de imágenes o tomándolas directamente de una cámara y hacer procesado en tiempo real. Después de esto, se explican cómo se ha implementado cada una de las funciones que son necesarias en el algoritmo para una mayor comprensión del código usado.

Glosario de abreviaturas

HW- Hardware

SW- Software

SoC- System on Chip

SDK- Software Development Kit

HLS- High Level Synthesis

FPGA- Field-Programmable Gate Array

PCA- Principal Component Analysis

VHDL- Lenguaje de descripción hardware de alto nivel

CPU- Unidad central del Procesamiento

SO- Sistema operativo

RTL- Register transfer level

LUT- Look Up Table

DSP- Digital Signal Procesor

BRAM- Block RAM

KPI- Key Performance Indicators

RMSE- raíz del error cuadrático medio

Matriz *U*- matriz de transformación para pasar a espacio con dimensionalidad reducida en PCA

Matriz *I*- matriz que acumula las imágenes originales que se usarán para conformar el fondo en PCA

Matriz *ψ*- matriz que contiene el valor medio de cada uno de los píxeles extraído de las imágenes de fondo en PCA

Matriz *A*- matriz que almacena el valor de las imágenes de fondo menos la media

Matriz *V*- matriz de autovectores de $A^t * A$

Matriz *φ*- matriz resultante de restar la media a la imagen recibida

Matriz *Φ*- proyección de la imagen recibida en espacio transformado

Matriz *x*- imagen recibida para procesar en la parte online de PCA

Matriz x_r- imagen reconstruida desde el espacio transformado

Matriz *MD*- matriz que acumula el error de reconstrucción píxel a píxel

1. Introducción y objetivos

Actualmente el mercado está demandando cada vez más aplicaciones en las que el tiempo de procesado de la información es crucial. Un campo donde este hecho queda patente es el de la visión artificial, donde hay que procesar gran cantidad de datos en muy poco tiempo para poder dar una salida lo más rápidamente posible. Por ello, han aparecido nuevos sistemas empotrados, denominados SoC, que permiten la integración, en una misma pastilla, de elementos tales como uno o varios microprocesadores, memoria, periféricos… Un elemento que se pueden encontrar en algunos de este tipo de sistemas es una unidad de lógica programable (FPGA). En estas, se puede programar Hardware a través de Software. Así se puede conseguir construir sistemas fácilmente modificables pero que cuentan con la rapidez de ser un circuito creado específicamente para una función. Lo que le da rapidez al procesamiento.

Una de las empresas líder en este tipo de SoC es Xilinx. Esta empresa fabrica chips en los que se incluyen uno o varios procesadores y una FPGA en un mismo encapsulado. Para poder usarlas y programarlas con facilidad se han creado una serie de programas, también creados por Xilinx, como son el Vivado, el SDK o el HLS. Estos permiten hacer un particionado en Software/Hardware de un algoritmo y así conseguir, usando las bondades de cada uno de los mundos, tiempos de ejecución mucho menores.

Desde 2015, existe un programa que permite hacer este particionado de forma sencilla y desde un mismo programa. Contrasta con otras propuestas previas donde como mínimo había que usar 2 programas: Vivado y SDK. Además en este nuevo programa se escribe todo el algoritmo en lenguajes de programación de alto nivel, como por ejemplo C o C++ donde seleccionando una opción, se puede pasar una función de Software a Hardware de forma sencilla e intuitiva. No es necesario crear todos los buses de interconexión entre las dos partes ni saber lenguajes de descripción HW como puede ser VHDL. De esta forma, el tiempo de desarrollo de aplicaciones en las que se vaya a usar este particionado, se reduce considerablemente, ya que se puede hacer todo en software y comprobar su funcionamiento de forma rápida. Luego para pasar bloques a Hardware y configurar todos las conexiones entre cada una de las partes solo hay que seleccionar una opción y, automáticamente, el programa creará todo. Para indicar al programa el cómo debe implementar en Hardware las funciones seleccionadas o qué tipo de bus usar para transmitir los datos de una a otra parte del sistema se puede recurrir al uso de sentencias pragma. Éstas son ampliamente usadas en lenguajes de programación y permiten intercalar órdenes a las herramientas como compiladores y en este caso sintetizadores, entre el código desarrollado.

Con este TFG se pretende conseguir tener una visión bastante amplia de cómo operar con esta herramienta para conseguir que los proyectos que se creen se particionen en HW/SW de la forma que el diseñador vea más conveniente. Además de poder tener indicadores, también llamados KPI, que permitan decidir cómo de bueno es un particionado de un proyecto. También se querrá ver como la decisión de introducir o no una función en Hardware o dejarla en Software puede influir en la aceleración final del algoritmo y en qué medida. Al igual que puede afectar la forma de implementar una función. Otro aspecto que se ilustrará en este trabajo es el hecho de ver si el coste, en este

caso se basará en recursos consumidos, de introducir una nueva función justifica la aceleración que introduce.

Para cumplir con todo esto, se ha divido en varios capítulos este trabajo. En el capítulo 2, se dará una contextualización de qué ámbitos y qué tipo de aplicaciones está siendo usada en la actualidad esta nueva herramienta. En el 3 y el 4, se hará una explicación de todo el HW y SW que se va a usar en este trabajo. En el caso de los programas se explicará para qué se ha usado cada uno de ellos y cuáles son sus principales funcionalidades. Para el Hardware se ha optado por explicar las características que presentan, sobre todo, aquellas que se necesitarán para el desarrollo del trabajo. Como puede ser el número de LUT que tiene la FPGA o los puertos I/O más interesantes para el trabajo.

Tras estos capítulos de presentación de todo lo que se va a usar y de presentación del contexto actual de la herramienta, se pasa a los capítulos en los que se desarrollará propiamente el trabajo técnico desarrollado. En el capítulo 5, a modo de tutorial, se expondrán los pasos más importantes para poder realizar un proyecto en esta herramienta. Además de estos primeros pasos, se explicará como introducir un SO. En este caso, se usará PetaLinux, ya que las especificaciones del chip no permiten cargar un Linux completo. Además de que este es más liviano y tiene toda la funcionalidad que se necesita en el trabajo, lo que hará que la sobrecarga del sistema por meter un SO sea la menor posible. También se mostrará la forma de incluir las librerías de funciones de visión artificial (OpenCV y xfOpenCV). En el capítulo 6, se procederá a realizar el particionado del algoritmo PCA, usado como ejemplo para poder ver con mayor detalle la forma de trabajar en esta herramienta. Al final de este se dará una serie de resultados medidos de tiempo de proceso para ilustrar lo que es capaz de realizar este programa.

Luego habrá un capítulo de conclusiones, capítulo 7. En él también se incluirán futuras líneas de trabajo para hacer el algoritmo más liviano y que se pueda ejecutar con mayor velocidad. Para terminar, se encuentra el capítulo de Planos, donde se hará una explicación de la forma de implementar cada función en código para su mejor comprensión.

Para concluir, se hará un presupuesto del gasto que podría suponer realizar este proyecto. Además se incluye un pliego de condiciones necesarias para poder repetir el proyecto sin que haya problemas de tener versiones que no soporten funcionalidades aquí usadas o por tener un PC con unas características que hagan casi imposible la ejecución de los programas usados.

2. Estado del arte

Los dispositivos SoC se usan en gran diversidad de campos ya que permiten poder integrar en un solo chip una FPGA, varias CPU y más periféricos. Lo que brinda la oportunidad de tener gran versatilidad en cuanto a las aplicaciones donde se pueden usar. Además permiten tiempos de cómputo reducido para aplicaciones de gran carga computacional, como puede ser la visión artificial o procesado de imagen. Debido a que pueden usar un procesamiento en paralelo entre CPU y FPGA. Una aplicación concreta sería la videovigilancia ya que, al tener está gran velocidad de cómputo, se puede grabar y procesar la imagen en mayor calidad y más FPS en tiempo real. Lo que hace que sea más fácil poder identificar personas o elementos en una escena. Con este trabajo se pretende conseguir que la aceleración del procesado sea la máxima de una forma sencilla, para así conseguir que esta demanda de cada vez más calidad de imagen y mayor número de imágenes por segundo se pueda afrontar sin que tiempo de cómputo necesario se dispare.

Debido a que SDSoC es un programa que ha surgido relativamente hace poco (julio del 2015) y que no tenía precedentes de ningún programa similar con estas características tan bondadosas, la aplicación no ha conseguido aún extenderse en gran medida su uso en trabajos o artículos. Por esto el número de trabajos encontrados en los que se hace uso de este programa es reducido.

Los trabajos previos encontrados se centran en ver cómo de eficiente es la planificación de las funciones HW que hace este nuevo programa con respecto a la que se hace forma manual [Suriano, 2017]. Donde se concluye que la planificación de SDSoC es algo peor que la manual, pero se ahorra mucho tiempo y esfuerzo con este nuevo programa. También hay líneas de trabajo orientadas en ver si la herramienta permite que los programas realicen una reprogramación de la FPGA durante la ejecución del programa [Kalb, 2016]. Así se puede tener varias funciones que estén aceleradas por HW pero sin la necesidad de que estén todas programadas a la vez y así ahorrar recursos y se consiguen optimizaciones mejores, ya que en un instante concreto tienes pocas funciones incluidas dentro de la placa lo que permite hacer los circuitos lo más comprimidos posibles y no tener rutas muy largas por donde se tienen que mover los datos que procesa la función implementada.

También se han encontrado proyectos de fin de grado en los que se trabaja con esta herramienta. Como por ejemplo [Torres,2017] que consigue detectar tumores cerebrales con el uso de imágenes hiper espectrales. Para ello, usa técnicas de PCA y SVM para reducir la dimensionalidad y poder clasificar. O [López, 2017] que implementa un algoritmo de compresión de imágenes de baja latencia. En todos estos casos comentan el uso de SDSoC y extraen datos, pero no exponen unos primeros pasos para su uso y consejos útiles para hacer un buen uso de ella. Por lo que aquí, se pretenderá dar estás claves iniciales para que el aprendizaje del uso de esta nueva herramienta sea lo más simple posible.

En cuanto al uso de SoC para el procesamiento del algoritmo PCA, se puede observar que se usa para reducir la complejidad de los datos a procesar y que la cantidad de información con la que hay que trabajar sea menor. Esto se puede ver en [Torres, 2017] por ejemplo. Para detectar nuevos objetos en un fondo también se han encontrado trabajos

[Bravo,2007], aunque todo el procesado se lleva a cabo en la parte HW. Pero lo que aquí se pretende es ir mucho más allá porque se va a hacer uso de un particionado HW/SW, lo que permitirá paralelizar tareas. Además se va a introducir un sistema operativo y se va a hacer uso de las funciones de OpenCV. De esta forma se podrá trabajar con ficheros. Lo que permitirá guardar los resultados para poder visualizarlo posteriormente. Además de poder introducir al sistema unas imágenes previamente tomadas para que las procese.

3. Software usado en el trabajo

3.1 SDSoC

Es una herramienta creada por la empresa Xilinx.Inc que permite que el desarrollo de aplicaciones para un SoC, en las que se realizará un particionado en parte Hardware y Software, se puede acelerar en gran medida. Ya que, pulsando en una opción, se puede mover una función de SW a HW e implementar todo el sincronismo y canales de comunicación entre las dos partes de forma automática y transparente para el usuario. De esta forma, se ahorra el procedimiento de crear en HLS un *IP core* concreto para esta aplicación, luego importarlo y unirlo al procesador con Vivado y, por último, a través de SDK, conseguir que el microprocesador se comunique con el HW para dar las instrucciones de cuándo tiene que realizar la función implementada en la FPGA y programar las acciones que tiene que realizar el micro. Esto ayuda a que el proceso de diseño de una aplicación destinada a SoC sea mucho más rápido, permitiendo así conseguir reducir el "*Time to Market*" y que las empresas sean más competitivas. Ya que no hay que hacer tantos procesos de comprobación de que todo funciona de forma correcta y la detección de errores en el diseño, al crearse todo desde un mismo programa, es mucho más sencilla. Además, al estar todo el proceso de paso de una función de Software a Hardware automatizado, es muy complicado producir fallos en la implementación en HW del sistema, ya que de esto se encarga por completo el SDSoC. Otra ventaja es que la forma de solventar los posibles errores que puedan surgir en una aplicación durante su uso se pueda solventar con mayor celeridad y sin tener que dedicar mucho tiempo y esfuerzo. Lo que implica ahorro de costes de mantenimiento de la aplicación y que la incorporación de futuras mejoras sea más fácil de realizar.

Los lenguajes soportados por esta aplicación son C, C++ y OpenCL. También permite hacer uso de las librerías de código abierto OpenCV. Pero para poder usarlas hay que indicarle al programa donde se encuentran las librerías en el sistema de archivos para que el linker pueda funcionar de forma correcta. Otra biblioteca de funciones que son interesantes usar en SDSoC son las librerías de funciones xfOpenCV. Creadas por Xillinx y están diseñadas para optimizar el rendimiento y consumo de recursos de las funciones de OpenCV al introducirlas en Hardware. Estas funciones, en su interior, usan funciones de OpenCV estándar. Por lo que si se quieren usar, hay que incluir las cabeceras de los dos conjuntos de funciones: las de xfOpenCV y las de OpenCV. De momento, hay un pequeño conjunto del total de funciones optimizadas pero con cada versión van aumentando en número.

Está construido sobre Eclipse. Para hacer la implementación de la parte Hardware invoca a los programas de Vivado y HLS desarrollados por la misma empresa. En cambio, para compilar los programas que se ejecutarán en la parte Software, hace uso del compilador que ofrece Eclipse. En cuanto a las opciones de sistema operativo que permite para construir los proyectos que se desarrollan en ella, acepta *standalone* (sin sistema operativo, también llamado *baremetal*)**,** Linux (se usa una versión reducida de este sistema, llamada PetaLinux, para que pueda entrar en la RAM de la placa) o un *FreeRTOS* (sistema de tiempo real). Al construir un proyecto, el programa genera tanto el *bitstream* necesario para programar la parte Hardware como el programa que correrá en la parte del procesador. Si además se ha elegido un sistema operativo sobre el que se desea que corra

el programa compilado, genera todos los archivos necesarios para que al arrancar la placa desde la SD se cargue todo el sistema operativo que se ha seleccionado.

Otro aspecto importante de SDSoC es el uso de pragmas para poder indicar al compilador cómo debe hacer la asignación de recursos en una parte del código, por ejemplo hacer un *pipeline* de un bucle, o cómo realizar las conexiones de la parte Hardware con memoria compartida o con el exterior, por ejemplo usar una cola FIFO para leer los datos de un array muy grande. Esta puede afectar a una operación, una parte de una función (p.e un *for*) o una función entera. De esta forma, se consigue reducir la latencia del procesado hardware a costa de aumentar el uso de recursos de la FPGA. Ahora es cuestión del diseñador ver si el aumento de recursos está justificado con la reducción de tiempo que se consigue. Hay de dos tipos: los propios del programa HLS (son de la forma #pragma HLS…) y los específicos de SDSoC (#pragma SDS…). Los de HLS se explicarán con detalle en la explicación del programa en cuestión. En cuanto a los de SDSoC, hay que comentar que son menos numerosos que los de HLS, ya que este programa lleva menos tiempo en uso y han tenido menos tiempo para su desarrollo. Se suelen colocar precediendo a la declaración de la función donde van a afectar y su efecto suele ser sobre una variable de esa función. Se dividen en varios grupos según su utilidad: están los de patrones de acceso (para indicar si se accede a los datos de forma secuencial o aleatoria), para ver si se copian los datos de memoria o se trabaja sobre ellos directamente (data copy zero_copy), para colocarlo en posiciones de memoria física continua o no continua (mem_attribute), el tipo de tecnología que se quiera para mover los datos (FIFO, DMA), para interfaces con memoria externa (data sys_port), la longitud del buffer hardware (data buffer_depth), para ejecución de funciones asíncronas (async, wait) y otras que sirven para hacer seguimiento de la parte HW y de la SW (trace), generar varias implementaciones de la misma función (resource) o generar varios *bitstream* uno para cada parte del código y que se carguen de forma automática en tiempo de proceso (partition). En este trabajo se usarán las de patrones de acceso y las de no copiar desde memoria de programa a la parte HW.

Aparte de los pragma, hay también una serie de funciones que se usarán para asignar memoria dinámica en posiciones de memoria física contigua. Las usadas en este trabajo son sds_alloc y sds_free. Para reservar y liberar memoria respectivamente.

Cabe destacar que también permite acceder al diseño por bloques del sistema que ha generado a partir del código y las sentencias pragma que se le ha impuesto al compilador del programa. Para ello, hay que abrir el archivo .xpr, situado dentro de la carpeta *<configuración de construcción del proyecto: Release o Debug>/_sds/p0/vivado/prj,* desde Vivado.

También cuenta con una herramienta de perfilado, llamada *TCF profiler*, que se usa para ver qué funciones usan más la CPU y luego poder elegir mejor las funciones que son mejores candidatas para ser aceleradas. Para poder analizar estos datos con claridad hay que tener en cuenta que el perfilador nos devuelve dos datos: *Inlcusive* (muestra el porcentaje de tiempo que el programa está dentro de una función y de las funciones a las que llama esta) y *Exclusive* (solo contempla el porcentaje de tiempo que está dentro de la función a la que hace referencia, sin contar el tiempo dentro de las funciones a las que llama en su interior). La forma de medir el tiempo es generando muestras de forma

periódica y en cada muestra mira dónde está el programa ejecutándose. O sea que en verdad lo que se tiene es porcentaje de muestras dentro de una función. Además de decir la función a la que hace referencia el dato de porcentaje de tiempo dentro de ella, también te muestra en qué archivo se encuentra y la línea dentro de este dónde empieza la declaración de la función. Para mostrar esto último la función tiene que haber sido creada por el usuario o estar dentro de las librerías que se incluyen al proyecto.

Algo que también se usará bastante es el estimador de recursos y ejecución. Esta opción se activa en las opciones generales del proyecto (*Estimate Performance* dentro del archivo "project.sdx"). Esta utilidad hace una estimación, antes de hacer la implementación en HW, de cuánto va a tardar en procesar el programa si solo se usa SW o si se usa SW y la aceleración de alguna función a través de HW. Al presentarte estos dos datos lo acompaña de un tercero, que es de gran interés para este estudio: el *Speed-Up*. Estos datos se muestran para todo el algoritmo, y de cada una de las funciones que se ha decidido acelerar. Además viene acompañado de una estimación del uso de recursos dentro de la FPGA que van a hacer las funciones aceleradas. Esta estimación es general y no particularizada para cada una de las funciones que se acelera. Muy útil para ver si una función interesa o no acelerarla (si aceleración mayor que 1 acelera el HW, si es menor mejor procesado SW) y ver también el consumo de recursos que se va teniendo según se van incluyendo más funciones. No es bueno llegar a niveles de ocupación cercanos al 100%, ya que la implementación tiene que hacerse por conexiones más largas lo que hace que el tiempo de proceso sea mayor y la aceleración de cada función menor. Además de suponer un mayor consumo de energía, pero eso en este trabajo no es algo que se vaya a valorar como motivo para seguir unas estrategias en vez de otras. Este estimador funciona tanto para proyectos *standalone* como para aquellos en los que se haga uso de un sistema operativo, ya sea PetaLinux o FreeRTOS.

También incluye un analizador de eventos que permite ver cuánto tardan en moverse los datos de un lado a otro del sistema y cuánto tardan en ejecutarse cada una de las operaciones del programa. Al igual que se puede ver cuales se paralelizan para conseguir mayor velocidad de proceso.

Esto viene bien para cuando se está haciendo un proceso de Debug, tiene una pestaña para esta finalidad, y se quiere ver qué está fallando si es un problemas de tiempos. Este depurador permite ir paso a paso viendo qué ocurre en cada instrucción que se ejecuta y ver cómo van cambiando las distintas variables que están implicadas en cada acción. Desde está pantalla es desde donde se lanza la herramienta de perfilado nombrada anteriormente (*TCF Profiler*). Las formas más comunes de depurar son: en un emulador sin descargar en placa, descargando en placa y para seguir los eventos que pasan en el sistema. Para aplicaciones *standalone*, se puede hacer una depuración en placa a través de usar un JTAG (en este caso es un cable USB a micro USB que se conecta a un puerto indicado a tal efecto) y la placa en modo arranque a través de JTAG o usar el emulador QEMU, que permite ver cada una de las señales implicadas en el diseño en una pantalla de formas de ondas de Vivado. Para el caso de usar un sistema operativo hay que conectar la placa al ordenador a través de una conexión Ethernet y configurar el TCF Agent para que se conecte a la dirección asignada a la placa. Una vez hecho esto, la forma de depurar es la misma que si fuese sin SO y se estuviese usando un JTAG.

Este programa está muy ligado a HLS, ya que si se quiere hacer una optimización mejor de las funciones que se quieren acelerar, hay que recurrir a él. La forma de exportar una función a acelerar a HLS desde SDSoC es seleccionarla y pulsar la opción  y se hará la exportación. Una vez en él, se hace el proceso de optimización igual que se hacía al solo usar HLS. Por último recordar de poner las sentencias pragma en el código y guardar el archivo. Al volver a SDSoC esos cambios estarán reflejados y si se lanza un estimador de ejecución se verá que los resultados sobre uso de recursos y tiempo cambian.

Para concluir con la presentación de las virtudes de este programa cabe destacar que es capaz de crear plataformas Hardware personalizadas. Además, indicándole una serie parámetros, te genera un FSBL para poder arrancar el sistema operativo en la placa de forma sencilla. De igual forma también se puede crear la imagen de arranque del sistema operativo de forma sencilla. Por último hay que indicar que permite programar la flash de la tarjeta para introducir los datos que uno desee.

3.2 HLS (High-Level Synthesis)

Esta herramienta de Xillinx permite generar, a partir de códigos en C, C++ o System C, *IP cores* que son bloques que se implementarán en HW para desarrollar una función concreta. Su ventaja principal es que permite crear diseños HW a muy alto nivel y no tener que emplear lenguajes de programación de más bajo nivel como VHDL, que pueden ser más difíciles de comprender o peores para encontrar errores. Además estos últimos tienen mucha menos flexibilidad a la hora de implementar algunas funciones, como puede ser el caso de bucles.

Esto permite hacer desarrollos más rápidos de bloques HW, aunque el proceso tenga más pasos. La forma de trabajar con este programa es la descrita a continuación. Primero hay que definir la función que va a realizar el nuevo bloque HW en código de alto nivel, por ejemplo C. Una vez hecho esto, se simula para ver que todo funciona como se esperaba. Con esta confirmación, se pasa a realizar la síntesis para obtener la versión HW de este bloque. En este punto se debe hacer una cosimulación para comprobar que la síntesis se ha hecho de forma correcta. Ahora ya se puede exportar el RTL para incluirlo en el proyecto de Vivado. Por lo que se puede observar de este proceso no es un programa que tenga un objetivo final que se consiga, sino que se usa como apoyo para hacer los proyectos, tanto de Vivado como de SDSoC. Se podría definir como una herramienta que crea bloques HW personalizados para usarlos en el desarrollo que se está realizando en otro programa.

Al realizar el proceso de síntesis, se puede ver la latencia de procesado estimada de bloque HW que se está construyendo antes de introducirlo en el sistema general. Además se puede estudiar el consumo de recursos que va a tener el nuevo bloque a través de un *report* que devuelve el programa. En este también se puede ver si la propagación de la señal dentro de un circuito es suficientemente rápida para que se puedan tener los datos de salida antes de que llegue el siguiente flanco de reloj. En este mismo archivo también se muestra el tipo de interfaz que tendrá cada una de las salidas/entradas del *core* que se está generando.

Una forma de ver lo que está pasando dentro del *core* y llevar un control de la secuencia de operaciones que se están produciendo en el interior es con el *Analysis Perspective.* Permite ver cuántos recursos HW consume cada una de las partes que conforman el código que implementa la funcionalidad del bloque IP que se está creando. Además de cuándo se ejecuta cada una de estas operaciones y cuántos recursos consume cada parte del código usado. La forma de separar por partes el código es a través de la inclusión de etiquetas en las líneas de interés. Un ejemplo podría ser la línea donde comienza un bucle.

Uno de los puntos fuertes de este programa es el uso de sentencias pragma. Como se indica en la descripción de SDSoC, son instrucciones que indican al compilador cómo realizar la implementación en HW. Estas son más numerosas que las propias de SDSoC. Aunque SDSoC acepta todas las sentencias pragma que se introduzcan en el código y HLS solo las suyas. Estas suelen estar dentro de la función y suelen afectar a una parte del código, aunque también pueden afectar a la forma de trabajar con la variable a la que hacen referencia. Estas tienen utilidades como: *pipeline*, optimización o unroll de bucles, optimización de los arrays, empaquetado de estructuras (se pasa de una estructura a un vector grande y se ahorra memoria) y optimización del Kernel (latencia, recursos). El más usado en este trabajo será el de *pipeline*. Ya que las funciones a acelerar son, básicamente, bucles que realizan una acción concreta. Otro que también se usa bastante es el de *Interface* para poder elegir cómo debe de ser la conexión con el exterior de cada una de las entradas/salidas del IP *core*. Pudiendo elegir entre cualquier tipo de AXI4 o un bus convencional. Y estas si suelen ir al inicio de la función.

Ya que las posibilidades de combinaciones de sentencias pragma que se puede usar son bastante elevadas, debido al gran número entre las que se puede elegir. Existe la posibilidad de comparar varias opciones en las que se aplica varios pragmas a distintas partes del código y se va viendo cómo se va modificando el uso de recursos y la latencia del bloque. Finalmente, se elegirá aquella que la relación recursos-latencia sea mejor.

Este programa correrá bajo SDSoC para hacer la implementación Hardware de las funciones que se marquen a tal efecto. Además se podrá acceder a él a través de una opción diseñada a tal efecto para poder realizar una optimización personalizada de la forma en la que implementará el programa en Hardware la función que se desee. Este procedimiento se denominada *"Fine Tunning"*.

3.3 Vivado

Programa que engloba todo el desarrollo del diseño de un proceso para implementarlo en un SoC, desde la programación del Hardware hasta la creación de programas que corran en el micro y la comunicación entre las dos partes. Todo el desarrollo del software se realiza con la herramienta integrada en Vivado llamada SDK. Pudiendo realizar en cada paso del desarrollo una simulación para poder comprobar que todo va funcionando de la forma deseada.

Permite la opción de elegir los pines de la placa donde se va a conectar cada entrada/salida del sistema final. Además se puede elegir la tensión de cada uno de estos. Gracias a una interfaz gráfica se puede realizar el conexionado de cada uno de los bloques

que integran el diseño hardware que se está llevando a cabo. Además de elegir la forma en la que se va a unir el Hardware y el Software. Ya que se incluye un bloque en esta interfaz que representará al micro. Los bloques (*IP cores*) pueden crearse en el mismo Vivado con lenguajes de programación destinados a descripción de hardware, como puede ser VHDL o Verilog, o importarse desde HLS al proyecto. También hay que añadir que hay una gran variedad de bloques IP que vienen ya desarrollados por Xilinx para introducirlos y usarlos. Al igual que los de Xilinx, también se pueden usar aquellos que están creados por terceros con solo añadir al proyecto el repositorio donde están ubicados.

De la parte Software, se puede elegir activar y desactivar todos módulos a los que está conectado. Por ejemplo, timers o buses de comunicación con el exterior. Además de configurar los relojes que pueden servir de base de tiempo a la FPGA y la velocidad a la que trabaja la CPU.

Una vez generado el *bitstream*, se puede descargar en placa y probar el funcionamiento en ella. Otra función muy interesante es la del uso del analizador lógico de señales para poder ver el tiempo que tarda cada elemento en realizar su función y cómo se va propagando la información a lo largo de los distintos bloques del diseño.

Los seis pasos principales seguidos por este programa son: integrador de bloques IP, simulación, análisis RTL, síntesis, implementación y programación (generación del *bitstream*). En cada uno de ellos se puede ver un esquemático de como se ha generado el diseño en ese punto. En síntesis e implementación está la opción de ver informes de temporización, uso de recursos y añadir restricciones *(constraints)* temporales para que el diseño cumpla con los requisitos temporales que están impuestos al sistema.

4. Hardware usado en el trabajo

4.1 Zedboard

La placa sobre la que va a correr el algoritmo desarrollado durante este TFG y así poder ver la aceleración en el tiempo de proceso entre solo software y el particionado será la Zedboard de Avnet. Esta placa de desarrollo tiene como SoC el XC7Z020-1CLG484 de la familia Zynq-7000 de Xilinx. Se puede programar con JTAG, con QUAD-SPI o a través de una SD. Los osciladores de la placa son de 100MHz para la parte lógica (PL) y de 33.333MHz para la parte software (PS). Estos se pueden ajustar con un PLL interno. En el caso de la PS puede llegar hasta los 667MHz. Para la parte lógica se puede usar o el oscilador proporcionado por la placa o uno de los proporcionados por la parte software.

La parte software está formada por 2 procesadores ARM Cortex-A9. Mientras que la parte de lógica programable está compuesta por: 220 DSP, 140 bloques de BRAM, 53.200 LUT y 106.400 FF. En cuanto al bloque de comunicaciones con el exterior hay un puerto Ethernet, un puerto serie a través de USB, una ranura para SD, conectores para audio, puerto VGA y HDMI. Además de varios puertos para control de elementos externos. La placa cuenta con 8 switches, 10 LED y 9 pulsadores, dos de ellos para reset (1 parte hardware y otro para la software). Dispone de un pequeño panel OLED para poder mostrar una pequeña información por él. También tiene 512 MB de memoria RAM de tipo DDR3. Un esquema general de todo lo que se puede encontrar en esta placa se puede observar en la Figura 1.

En el caso de este TFG, se usará el modo de carga a través de la SD, usado para cargar en el arranque todo el SO que esta almacenado en ella y así poder operar con él. También usado cuando se tiene una versión definitiva del diseño a realizar y así no tener que conectar la placa al ordenador para cargar el programa cada vez que se enciende la tarjeta. El sistema operativo que se incluirá es el PetaLinux. Debido a que permite gran serie de funcionalidades de un sistema operativo con un uso de memoria reducido, recurso bastante escaso en la placa.

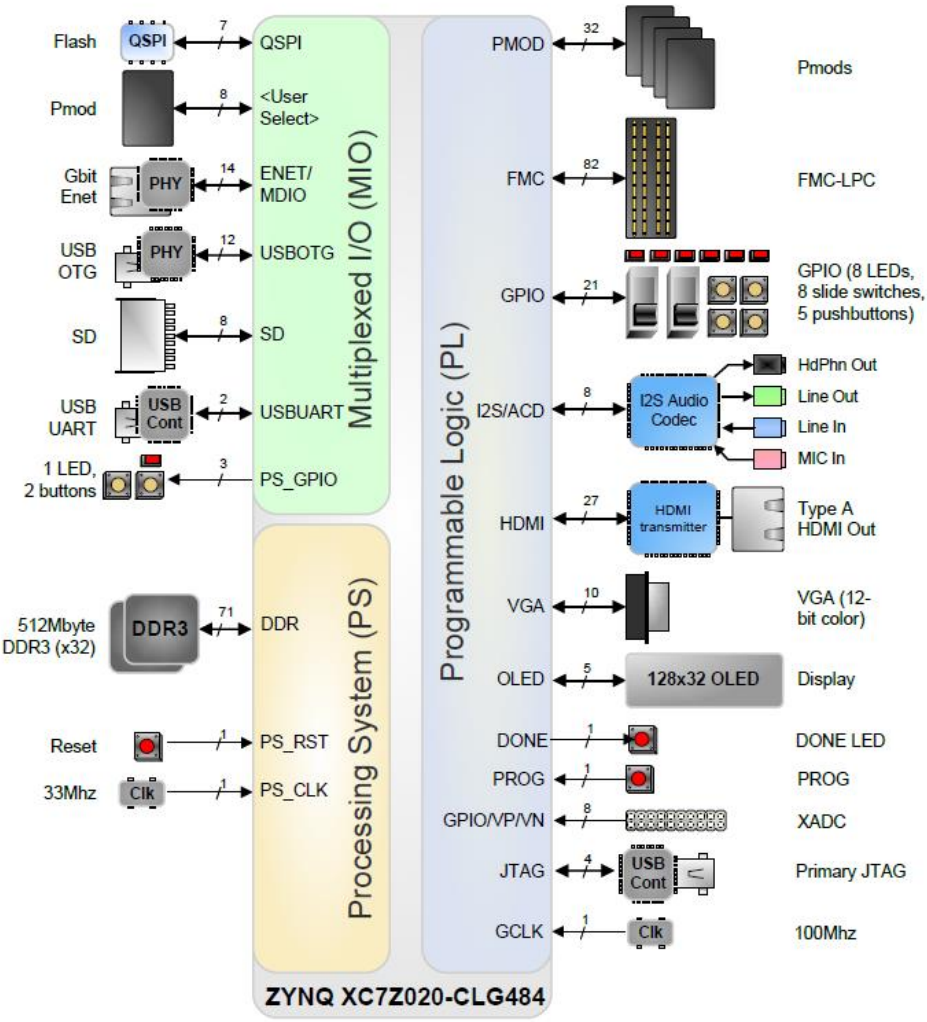


Figura 1: Diagrama de Bloques ZedBoard [Avnet,2014]

4.2 Zynq 7000 [1]

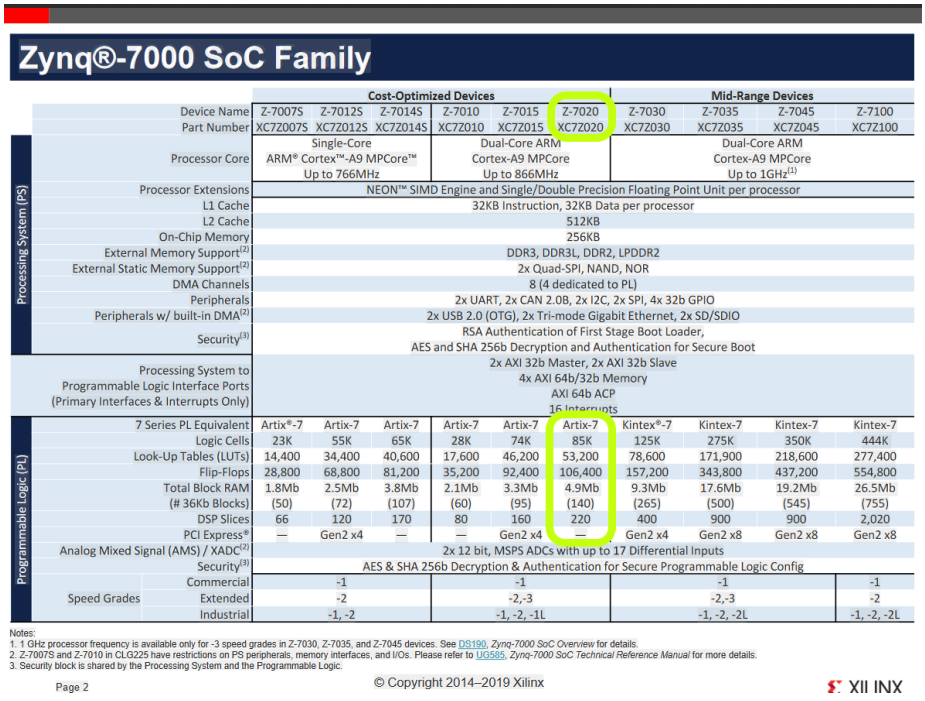
El núcleo central de esta placa de desarrollo es la pastilla de la familia Zynq-7000 construida por Xilinx. Concretamente la XC7Z020-CLG484. Este SoC cuenta con dos partes diferenciadas: la parte de sistema de proceso y la de lógica programable.

En la parte de sistema de proceso, hay dos cores ARM Cortex A9 que trabajan de forma independiente. En cuanto a memoria, hay 32KB en L1 para instrucciones y 32 KB para datos de cada procesador, 512 KB de caché de nivel 2 y 256 KB de memoria. Cuenta con periféricos integrados como el DMA, control del JTAG o timers. Todo este bloque está conectado al resto de elementos del chip y con el exterior de este a través de interconexión AMBA, buses AXI. También hay controlador de FLASH y de memoria DRAM multipuerto y varias conexiones de distinto tipo como puede ser UART o

[1] Extraído de [Xilinx, 2019]

Ethernet. Para interconectar con la parte de lógica programable se hace a través de buses AXI (de propósito general o de alto rendimiento). Al que hay que incluir un bloque que incluye seguridad entre las dos partes. La frecuencia de reloj máxima es de 866 MHz y puede tener hasta 128 pines dedicados a periféricos. Contiene una unidad de procesamiento en coma flotante por cada núcleo.

En cuanto a la lógica programable, es una FGPA de la familia Artix-7 de Xilinx. Tiene bloques de entrada/salida estándar para conectarse con el exterior. En cuanto a elementos que programables que contiene hay: 85.000 celdas lógicas, 53.200 LUT (Look Up Table), 106.400 Flip-Flops, 220 DSP y 140 bloques de BRAM de 36Kb. La memoria BRAM total es de 4,9 Mb. El *speed grade* es de -1. Todo esto queda recogido en la siguiente imagen.



Zynq®-7000 SoC Family

		Cost-Optimized Devices						Mid-Range Devices			
	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Processing System (PS)	Processor Core	Single-Core ARM® Cortex™-A9 MPCore™ Up to 766MHz			Dual-Core ARM Cortex-A9 MPCore Up to 866MHz			Dual-Core ARM Cortex-A9 MPCore Up to 1GHz[1]			
	Processor Extensions	NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor									
	L1 Cache	32KB Instruction, 32KB Data per processor									
	L2 Cache	512KB									
	On-Chip Memory	256KB									
	External Memory Support[2]	DDR3, DDR3L, DDR2, LPDDR2									
	External Static Memory Support[2]	2x Quad-SPI, NAND, NOR									
	DMA Channels	8 (4 dedicated to PL)									
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO									
	Peripherals w/ built-in DMA[2]	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO									
	Security[3]	RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot									
	Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave, 4x AXI 64b/32b Memory, AXI 64b ACP, 16 Interrupts									
Programmable Logic (PL)	7 Series PL Equivalent	Artix®-7	Artix-7	Artix-7	Artix-7	Artix-7	Artix-7	Kintex®-7	Kintex-7	Kintex-7	Kintex-7
	Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Total Block RAM (# 36Kb Blocks)	1.8Mb (50)	2.5Mb (72)	3.8Mb (107)	2.1Mb (60)	3.3Mb (95)	4.9Mb (140)	9.3Mb (265)	17.6Mb (500)	19.2Mb (545)	26.5Mb (755)
	DSP Slices	66	120	170	80	160	220	400	900	900	2,020
	PCI Express®	—	Gen2 x4	—	—	Gen2 x4	—	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC[2]	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security[3]	AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config									
	Speed Grades Commercial	-1			-1			-1			-1
	Speed Grades Extended	-2			-2,-3			-2,-3			-2
	Speed Grades Industrial	-1, -2			-1, -2, -1L			-1, -2, -2L			-1, -2, -2L

Notes:
1. 1 GHz processor frequency is available only for -3 speed grades in Z-7030, Z-7035, and Z-7045 devices. See DS190, *Zynq-7000 SoC Overview* for details.
2. Z-7007S and Z-7010 in CLG225 have restrictions on PS peripherals, memory interfaces, and I/Os. Please refer to UG585, *Zynq-7000 SoC Technical Reference Manual* for more details.
3. Security block is shared by the Processing System and the Programmable Logic.

Page 2

XILINX

Figura 2: Características XC7Z020

5. Introducción a la herramienta SDSoC

5.1 Primeros pasos en SDSoC

Como es una herramienta y una metodología de trabajo bastante reciente, se ha optado por realizar una pequeña introducción al programa para dar unas pautas que hay que seguir para crear un primer proyecto e indicar cuáles son las herramientas más útiles que se pueden encontrar dentro de él. El entorno de trabajo resulta familiar ya que está basado en Eclipse. Muy usado para desarrollar código en lenguajes de programación software, como por ejemplo C. Al igual que en este, hay que indicar un directorio de trabajo donde se guardarán todos los archivos realizados durante el desarrollo del programa.

Tras ello, habrá que irse a la pestaña "*File->New*" y crear un nuevo proyecto. En el caso del TFG y de todos los proyectos en los que se quiera desarrollar un diseño que, a partir de un código en C/C++, se implemente una solución en la que se haga uso de un SoC para ejecutar el algoritmo desarrollado, se deberá pulsar en la opción *SDx Aplication Project*. Después habrá que seleccionar la placa sobre la que se va a implementar el algoritmo que se desarrolle, en el caso de este trabajo la ZedBoard (indicada como zed en el programa). Se elegirá si se quiere que corra con sistema operativo (*FreeRTOS* o *Linux*) o sin él (*Standalone*). Por último se puede elegir entre unos cuantos proyectos base que ofrece Xillinx o empezar con una aplicación vacía. Tras aceptar y esperar a que el proyecto se cree, aparece un entorno de trabajo como el mostrado en la Figura 3.

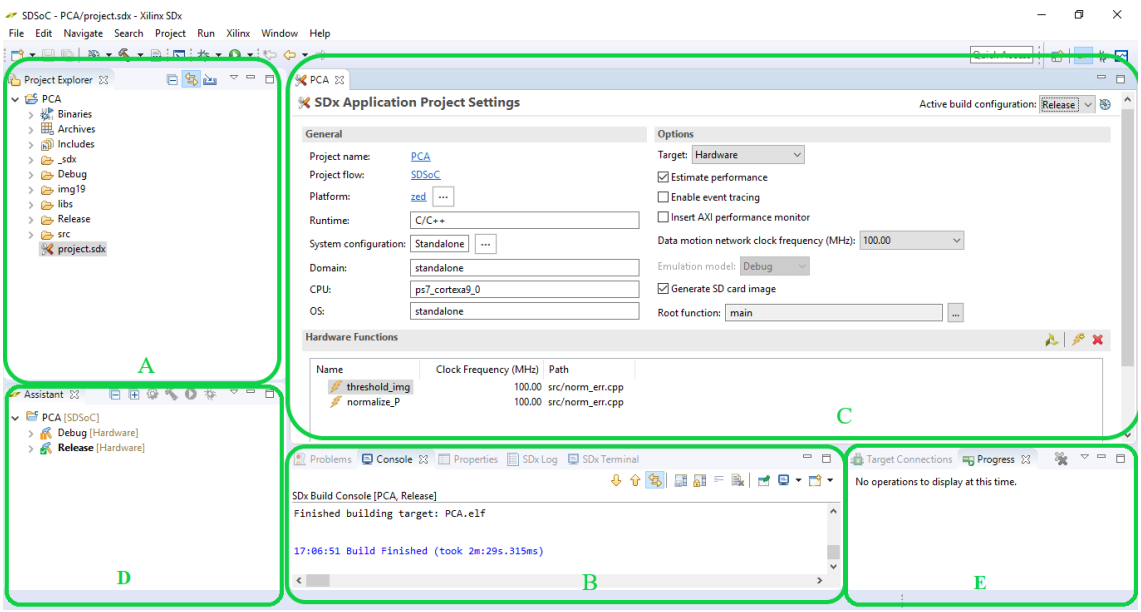


Figura 3: Inicio programa SDSoC

Se pueden diferenciar cinco bloques. El bloque A es el correspondiente a explorador de archivos, en él se podrá ver en todo momento todos los archivos que hay añadidos al proyecto como todas las carpetas en las que el programa busca librerías y archivos de cabecera cuando se ejecuta un proceso de compilación o de "*linkado*" para

generar el archivo ejecutable. Además tiene un desplegable donde se puede ver todos los ficheros ejecutables (.elf) que tiene en el proyecto.

El bloque B representa la consola. En ella se podrá ir viendo todo el proceso de compilación del diseño y, en el caso de que los haya, se mostrarán los errores y los *warnings* que puedan surgir a la hora de compilar la aplicación.

La parte D se emplea para ver si la compilación de todas las funciones aceleradas ha sido correcta. Estas están separadas según el modo de construcción configurado. Pudiendo ver los modos en los que la construcción del proyecto ha sido totalmente correcta y en cuál hay problemas. Dentro de cada función acelerada hay un informe de HLS que indica el consumo de recursos y la latencia de la función acelerada. Es el mismo que devuelve HLS al hacer la síntesis de una función en C a RTL. Además se puede abrir un informe de cómo es el movimiento de datos dentro de la red creada en la parte HW (*Data Motion Network Report)* y el registro de compilación. Este alamcena todo lo ocurrido durante la compilación del programa y puede verse en él los errores y *warnings* que se hayan producido en la compilación.

En las ventana E, se encuentra la pestaña de control de proceso. En ella, se irá indicando cómo de avanzados están los procesos que se estén ejecutando en cada momento en el programa y la cola de tareas que hay por realizar. La otra pestaña que tiene es la que se configura los servidores que controlarán los procesos de depuración. O sea depuración con o sin SO y el emulador QEMU.

Por último, en la región C se podrá ver y escribir todos los ficheros necesarios para la creación del proyecto. En el caso de la Figura 3, se muestra la vista de abrir el archivo “proyect.sdx”, que es el archivo con el que se controlan todas las características esenciales del diseño que se está realizando. En este, se puede configurar si se quiere que el proyecto tenga o no SO con solo cambiar una opción. También se usa para elegir si, al construir el proyecto, se creen los archivos necesarios para hacer funcionar un SO en la placa (opción *Generate SD card image*). Otra opción muy usada es activar el estimador de tiempo de proceso y recursos consumidos de las funciones indicadas para acelerar. Además puedes activar el seguidor de eventos (*event tracing*) del proyecto.

Una de las opciones más importantes que se debe configurar en este archivo de características generales del proyecto es el tipo de construcción del proyecto que se desea hacer. Pudiendo elegir entre modo *Debug* o *Release*. Esto permite usar un compilador con más o menos optimización, lo que supondrá más tiempo de construcción del ejecutable. Esto varía según el punto de desarrollo en el que se encuentre el proyecto. Siendo el más optimizado y cercano a la realidad el del modo *Release*. Con *Target* se indica si el proceso está destinado para probarlo en hardware o que haga unas simulaciones y así poder ver que todo funciona correctamente antes de pasarlo a placa.

Otra opción primordial para el desarrollo de la aplicación es la selección de las funciones SW que se van a pasar a HW. Esto se elige en esta pantalla. Al pulsar en el icono  se abre un desplegable donde podemos elegir, de todas las funciones creadas hasta el momento, las que se quieran pasar a HW. También se puede hacer desde el explorador de archivos seleccionando la opción *Toggle SW/HW* como se muestra en la Figura 4. De esta forma tan sencilla se le indica al programa a qué parte del SoC (PL o PS) tiene que llevar cada una de las funciones de la aplicación. Por defecto, las compila para procesarlas en la parte SW.

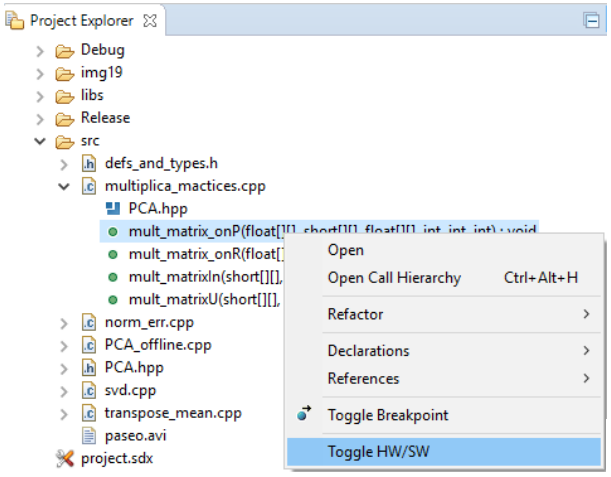


Figura 4: Cambio función HW/SW

Si se pulsa  , se abre HLS y se puede realizar un análisis de los recursos consumidos por la función que estuviese seleccionada a la hora de dar al botón. Además de poder introducir todas las sentencias pragma necesarias para poder llegar a la solución óptima entre consumo de recursos y tiempo de proceso. La  sirve para eliminar del hardware una de las funciones que se había añadido anteriormente.

Para concluir con el análisis del fichero de configuración .sdx, hay que decir que se puede ajustar también la velocidad de movimiento de los datos y la velocidad de reloj de cada una de las aplicaciones que se han seleccionado para acelerar. La modificación de uno de estos parámetros que se han explicado sobre este archivo obligará a la reconstrucción de todo el proyecto. Ya que se ha cambiado una característica que afecta a la construcción del proyecto resultante.

A la hora de crear código para introducir en la aplicación que se está desarrollando se puede elegir de otra parte del sistema de archivos (seleccionando la opción *Import* pulsando con el botón izquierdo sobre la carpeta donde se quiera incluir) o crear tantos ficheros de cabecera y de código como sean necesarios desde cero.

Una forma muy interesante de seleccionar las funciones candidatas a ser movidas a HW es a través del modo *Debug*. Para ello, una vez que el programa está creado y sin errores se pulsa en la icono de *Debug* () para cambiar a la pantalla del mismo nombre. La depuración del programa se puede hacer tanto para un programa *Standalone* como

para uno con SO, ya sea Linux o FreeRTOS. También se puede lanzar el emulador QEMU para ver cómo es la evolución de las señales del proyecto en el analizador lógico de Vivado. Pero esta opción no se usará en este trabajo.

Al abrirse la ventana de *Debug*, se pueden observar múltiples pantallas. Las que se usarán son: la ventana en la que se puede ver el código, la ventana de TCF Profiler, las variables y el explorador de archivos. En definitiva, se usarán las dos filas superiores. En la esquina superior izquierda se sitúa la ventana del explorador de archivos. Con ella, se podrá acceder a todos los archivos del proyecto y abrir el que se necesite en cada momento para ver el avance del código. Este se verá en la pestaña situada inmediatamente debajo de la recién comentada. Aquí se podrá ver el código y, con un sombreado verde, se indicará por la línea que va procesando el programa. Si se hace doble clic en el margen izquierdo se podrá poner un *breakpoint* en la línea que se desee para parar el proceso en ese punto. Para poder ver el valor de cada una de las variables que hay en el entorno de la función activa actual hay que fijarse en la ventana que se encuentra en la esquina superior derecha. Ahí se podrá ver tanto el valor actual de la variable como el tipo. Además permite ver el valor en octal, hexadecimal y decimal. Esta ventana tiene una pestaña que permite ver los *breakpoint* que se han puesto. Para acabar con el repaso de las ventanas más importantes para el proceso de depuración, hay que desplazarse a la derecha de la parte central de la pantalla. Ahí se encuentra la pestaña del TCF profiler, que se explicará con más detalle más adelante.

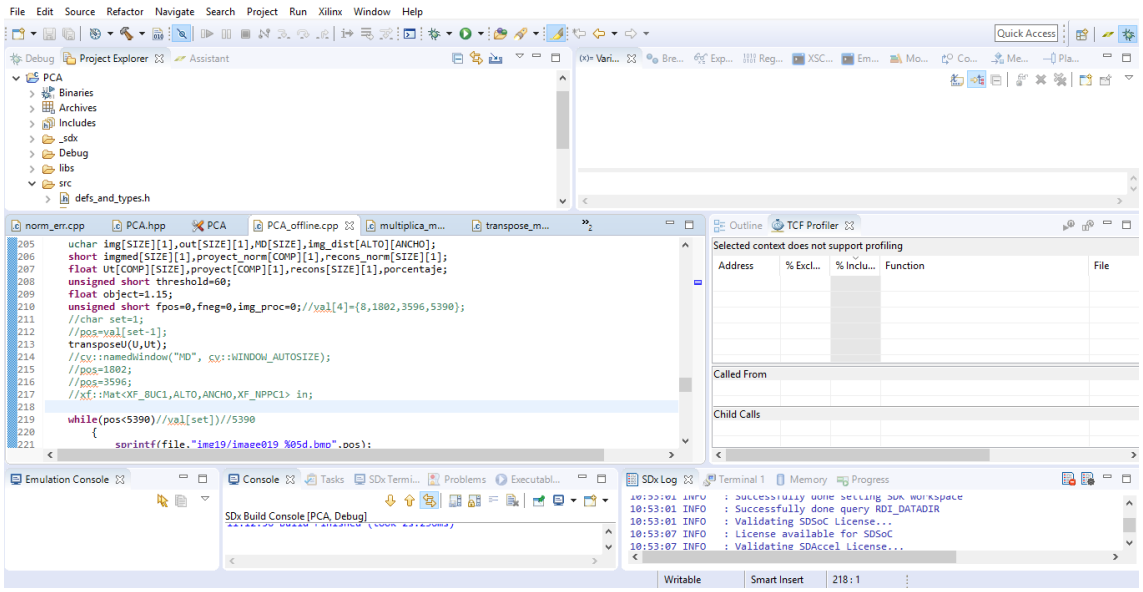


Figura 5: Pantalla debug de SDSoC

Gracias al *debugger* se pude ejecutar el proyecto paso a paso y así conseguir ver dónde están los posibles fallos del programa. Ya que se puede ir viendo, paso a paso, cómo evoluciona cada una de las variables. Pero lo que realmente hace interesante a este modo, aparte de solucionar problemas que pueda tener el código, es que permite ver cuánto tarda en ejecutarse cada función con ayuda del *TCF Profiler*. Este se encuentra en *Window->Show View->Other* y dentro del desplegable que se abre se elige *TCF Profiler*.

Esta herramienta permitirá tomar muestras mientras el programa se ejecuta y de esta forma ver qué funciones tardan más en procesarse en SW y cuáles menos. Por lo que las que más tarden serán las mejores candidatas para acelerar. Esto se debe a que son las que más fácil se va a conseguir mejorar y donde esta mejora puede provocar un impacto más grande en el rendimiento final de la aplicación a desarrollar. Porque una que tarda poco en ejecutarse en SW su ahorro de ciclos no va a ser tan significativo como una que tarda mucho en procesarse. De esta forma, se maximiza el valor del *speed-up* que se va a lograr. Objetivo primordial de usar esta metodología de trabajo de particionado HW/SW.

5.2 Trabajar con librerías de visión (OpenCV y xfOpenCV)

5.2.1 OpenCV y xfOpenCV[2]

Este programa permite incluir en el código que se está desarrollando funciones de las librerías de visión artificial de OpenCV. Las usadas en este trabajo son las que vienen incluidas dentro en el directorio donde se han instalado todos los programas de Xilinx, concretamente dentro de la carpeta de SDK. Estas son las mismas que se pueden encontrar en cualquier repositorio de internet donde estén disponibles estas librerías. En cambio, si se desea introducir en HW una función que contiene alguna función de visión artificial, es recomendable usarlas de las contenidas en la librería de funciones creada por Xilinx (xfOpenCV).

El motivo de tener que recurrir a esta librería de funciones es sencillo. Estas están pensadas para que sea un código fácilmente sintetizable y que su implementación sea óptima en cuanto a relación recursos-tiempo de proceso. Esto se consigue gracias a la inclusión de sentencias pragma que están perfectamente estudiadas. Hay que tener en cuenta que en su interior hacen uso de funciones de OpenCV, por lo que para usarlas hay que incluir las dos librerías.

Otra característica que le diferencia de las OpenCV estándar es que las matrices que se usarán se tienen que conocer sus dimensiones en tiempo de compilación. Además estas no pueden cambiar de tamaño en toda la ejecución del programa. Esto es debido a que en HW implementar asignación dinámica de memoria no es algo trivial, ya que los circuitos necesarios para realizar una operación no pueden estar cambiando continuamente de tamaño en función de las dimensiones de una variable que cambia en el transcurso de la ejecución del programa. También hay que tener en cuenta el tipo de dato que almacenara la matriz, para saber de qué tamaño serán los buses que se necesitarán usar para interconectar bloques IP. Esto se debe saber en tiempo de compilación para poder dimensionarlos correctamente a la hora de hacer la implementación HW. Hay que tener en cuenta que el HW permite paralelizar el procesado y se pueden estar ejecutando más de un píxel a la vez para mayor rapidez. Por lo que al definir una matriz que va a usar este tipo de funciones hay que indicar cuatro parámetros: tipo de datos a almacenar, altura de la imagen (número de filas), anchura de la imagen (número de columnas), y píxeles procesados en cada ciclo de reloj. De esta forma, la

[2] Toda la información relativa a las librerías xfOpenCV se ha extraído de [Xilinx, 2019-2].

declaración de una matriz sería: *xf::Mat <TYPE, ALTO, ANCHO, NPPC>*. La clase *xf* es lo que indica que se está trabajando con una función o una variable que está definida dentro de las librerías xfOpenCV. El número de píxeles que se pueden procesar por ciclo solo pueden ser tres valores: 1,2 o 8.

Como es tan poco versátil el HW a la hora de cambiar de tamaño una variable, si alguna función tiene como parámetro de entrada o salida una matriz de este tipo se debe indicar, de forma explícita, en su declaración las características que se han explicado en el párrafo anterior. Ya que solo esas serán las aceptadas como válidas para procesarse en la función. Si se quiere seleccionar para acelerar alguna función que contenga funciones xfOpenCV en su interior, es aconsejable seleccionar para acelerar de forma independiente cada una de estas funciones que no la función entera que incluye varias. De esta forma, se consiguen mejores resultados a la hora de llevar a cabo la implementación. Y, seleccionando la función más grande, puede dar problemas de compilación que de esta forma no saldrían.

Hay que comentar que el paso de información de una *cv::Mat* a una *xf::Mat* es bastante sencillo. Es cuestión de hacer un copyTo() a la matriz de xf desde la de cv. Para el proceso inverso se hace un copyFrom().

Las funciones que está disponibles en estas librerías no son todas las que se pueden encontrar en OpenCV, pero están las más comunes. Como por ejemplo *gaussianfilter o erode*. Aunque cada vez se van incluyendo más. Para la versión 2018.3, hay un total de 47 funciones soportadas. Y ya se han importado a esta librería de funciones las librerías de funciones de vídeo de HLS. En este trabajo, se valoró la opción de usarlas pero no suponían una mejora significativa en la aceleración general del proyecto. Por lo que se descartó su uso.

5.2.2 Pasos para incluir las librerías en un proyecto de SDSoC

Si se quiere trabajar con las librerías de visión de OpenCV o las desarrolladas por Xillinx (xfOpenCV) hay que realizar una serie de pasos extra para que el SDSoC puede generar el fichero ejecutable de forma correcta. Como en el proyecto desarrollado en el trabajo se hace uso de ellas, se va a explicar detalladamente como se tiene que hacer para que todo se construya de forma adecuada.

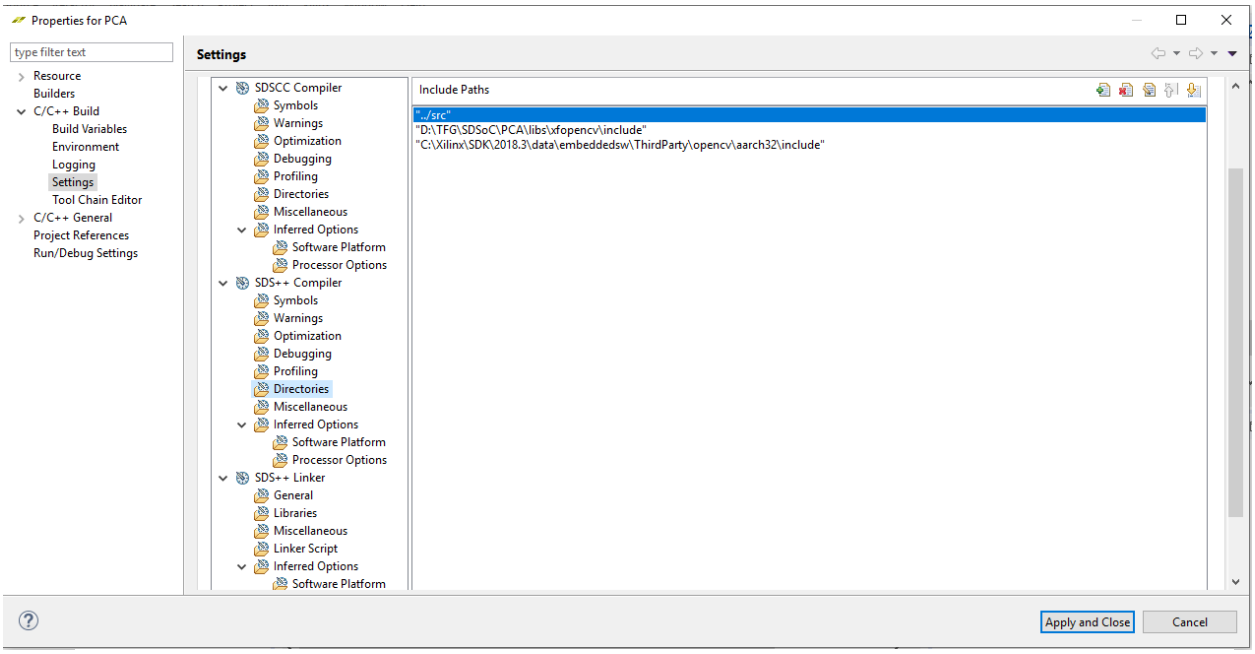


Figura 6: Pantalla mostrada al pulsar C/C++ Build->Settings en pantalla configuración.

Primero de todo hay que pulsar en la carpeta raíz del proyecto, en este caso tiene el nombre de "PCA", y seleccionar la opción *C/C++ Build Settings* para ir a la ventana de configuración que se muestra en la Figura 6. Dentro de ella se selecciona en la parte izquierda la opción *C/C++ Build->Setting* , como se puede ver ya en la figura.

En esta pestaña se pincha sobre la opción de *Tool Settings->SDS++ Compiler->Directories* y se añaden las rutas donde se encuentran las cabeceras de las funciones xfOpenCV y de las OpenCV. En el caso mostrado, se ha seleccionado las OpenCV que vienen instaladas con Xilinx. Para las xfOpenCV se ha elegido por añadirlas al proyecto y elegir el directorio donde te las añade el programa[3]. De esta forma, si se hacen alguna modificación sobre ellas solo se modifica la copia local al proyecto y no las originales. Cosa que es habitual al trabajar con ellas debido a su nula flexibilidad en cuanto al tamaño de los datos de entrada-salida de las funciones. Esta acción se debe repetir en el apartado *SDSCC Compiler* si los ficheros con los que se está trabajando están en C, en vez de en C++. Por último, hay que añadir en el apartado *Inferred Options->Software Platform* el flag "-hls-target 1", para indicar que los errores de compilación afectan al comportamiento de la parte hardware.

[3] Para poder hacer esto se debe pinchar en la pestaña Xilinx de la pantalla principal del programa (figura 2) y elegir al opción SDx Libraries. Dentro de ella, seleccionar la opción *Xillinx xfOpenCV Library* y pulsar la opción *Add to Project* y aceptar.

Una vez el compilador está totalmente configurado se pasa a configurar el *Linker*. Para ello, se selecciona el apartado *SDS++ Linker->Libraries.* La pantalla que se puede observar en este momento es como la Figura 7. En la parte superior se pondrán todas las librerías que son necesarias para poder unir todo el proyecto de forma correcta. En este caso son todas de OpenCV. Las xfOpenCV basta con poner dónde se encuentran las cabeceras para tener toda la compilación realizada correctamente. En la parte inferior se indican los directorios donde se tienen que buscar estas librerías (también se usan las que se instalan por defecto con Xilinx).

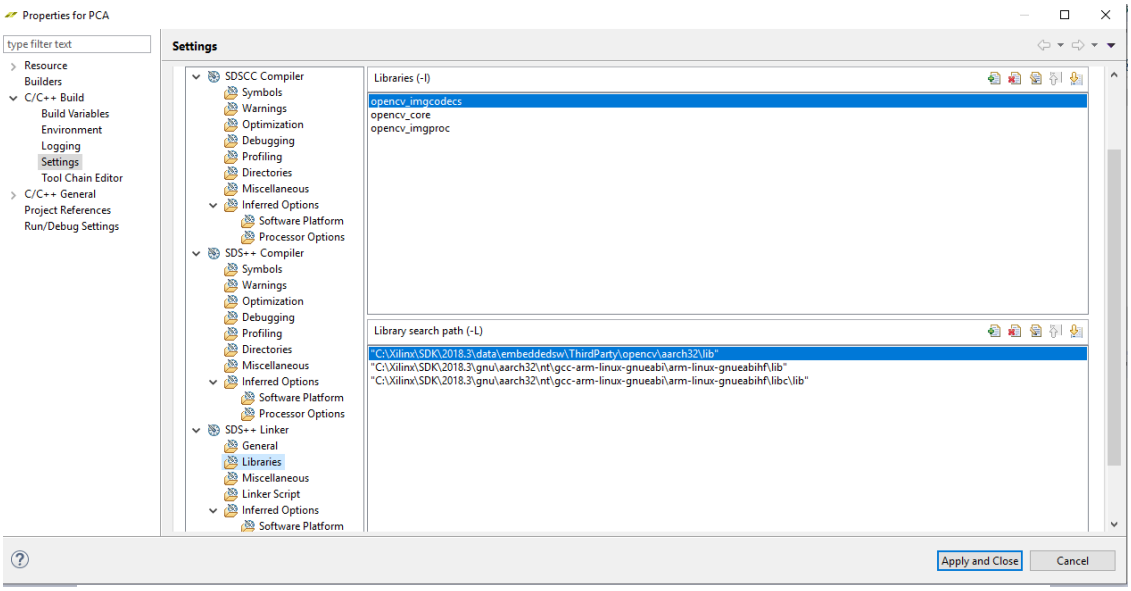


Figura 7: Pantalla para incluir librerías definidas por el usuario

Además hay incluidas otros dos directorios que son los que tienen unas librerías que necesitan las indicadas arriba para poderse compilar correctamente. Son librerías de c, c++ y de gcc básicamente.

Para concluir con la configuración, hay que irse al apartado *Miscellaneous* del *Linker* y escribir el siguiente comando: *-Wl,-rpath-link= "C:\Xilinx\SDK\2018.3\gnu\aarch32\nt\gcc-arm-linux-gnueabi\arm-linux-gnueabihf\lib",-rpath-link="C:\Xilinx\SDK\2018.3\gnu\aarch32\nt\gcc-arm-linux-gnueabi\arm-linux-gnueabihf\libc\lib"*. De esta forma, añado directorios para la que el proceso de "*linkado*" se pueda desarrollar sin problemas al añadir las OpenCV. Una vez realizado todos estos cambios ya se puede empezar a trabajar en la aplicación que se desee y que se pueda compilar sin que incluir funciones de OpenCV o xfOpenCV sea un impedimento.

Una vez escrito el programa que se quiere implementar en el SoC, se lanza el compilador presionando sobre el icono  . Una vez que tanto el compilador como el *linker* han concluido sin problemas, proceso que llevará un cierto tiempo (puede llegar a ser de 40-50 min de duración), ya está listo el ejecutable para poder correrlo en placa. El archivo .elf resultante tiene todo lo necesario para programar el microprocesador para que realice la aplicación deseada y las llamadas al HW para indicarle a este cuando tiene que operar y con qué datos.

5.3 Instalación de SO en SDSoC

Debido a que hay más de un sistema operativo que se puede elegir en este programa como se indica en el punto 3 de este subapartado, se va a explicar las principales características de cada una de las opciones y así justificar por qué se ha elegido Petalinux para la implementación de este trabajo.

5.3.1 Petalinux[4]

Petalinux es, básicamente, un Linux que se le han quitado opciones y características para dejarlo con lo fundamental. Y así poderlo usar en sistemas donde el almacenamiento y la RAM disponibles son reducidas. Como es el caso del SoC con el que se trabaja en el proyecto desarrollado. De esta forma se consigue adaptar las necesidades del SoC a los servicios que ofrece, ya que este sistema operativo esta creado por Xilinx para que sus integrados tengan todas las funcionalidades que necesitan, por ejemplo servidor web o drivers de los controladores de los elementos de la placa. Si se conectase algo exterior habría que instalar el driver a mano dentro el SO ya que este solo tiene los esenciales para funcionar y poder reducir al máximo el almacenamiento que ocupa.

Las principales características son: interfaz de línea de comandos, drivers para aplicaciones y dispositivos, una imagen del sistema que sirve como arranque, depuradores, herramientas de GCC, simulador QEMU, herramientas automáticas, se puede usar el Uboot como bootloader, soporta aplicaciones en C++ y da soporte al sistema de depuración del Xilinx.

El usuario tiene la opción de modificar el Kernel del sistema, el arranque y las aplicaciones de Linux. Los IP creados por Xilinx tienen total respaldo dentro de este sistema, ya que están incluidos los drivers necesarios para su correcto funcionamiento.

Se decidió optar por esta opción ya que el kernel es el de Linux limitado. Este sistema se tiene mejor conocimiento de su funcionamiento que el otro, ya que se conocen los comandos necesarios para realizar cada operación que se necesite. Además este permite una mayor flexibilidad en cuanto a introducir los drivers que se necesiten y, de los que ya están incluidos, permite alejarse del bajo nivel para centrarse en la aplicación de alto nivel que se está desarrollando. Además incluye un sistema de archivos que permite una mejor organización de los archivos que necesita el programa para funcionar, las imágenes del vídeo que se va a detectar la presencia de objetos.

5.3.2 FreeRTOS[5]

FreeRTOS es un sistema operativo de tiempo real que se usa para sistemas pequeños como un microcontrolador. Es robusto, se puede usar de manera gratuita en productos comerciales de manera segura y tiene buen manejo de los IP. Está totalmente documentado y cuenta con gran soporte para evitar cualquier riesgo a los usuarios.

Las características más llamativas de este sistema operativo son: es un sistema seguro, sigue en continuo desarrollo, ocupa muy poco espacio (<12KB), es muy simple, estable y muy escalable.

[4] Información extraída de [Xilinx]
[5] Información extraída de [FreeRTOS,2019]

No se eligió este porque no había problemas de que el PetaLinux no tuviese suficiente espacio en el sistema. No se necesitaba un control de tiempo real, ya que no hay restricciones de tiempo. Además, este no incluye los drives instalados ya por defecto en el otro sistema, por lo que habría que añadirlos a mano. No es tan versátil como el PetaLinux, que permite eliminar cosas que no se van a usar para hacerlo más liviano.

5.3.3 Pasos para integración de SO en proyecto de SDSoC

Como se ha optado por la opción de usar un SO, en este caso Petalinux, en la placa, se va a explicar, con unos sencillos pasos, la forma de tenerlo totalmente operativo dentro de la ZedBoard. Primero hay que tener en cuenta que se va a necesitar una tarjeta SD desde la que se arrancará la placa de desarrollo. En ella se deben de crear dos particiones: la primera con formato fat32 y el resto se formateará con el formato ext4. Para ello se ha usado el programa Gparted en Linux. En la primera, se guardará todo lo relativo al arranque del sistema operativo y el .elf de la aplicación. También se incluirá las librerías de OpenCV que se incluyeron en la construcción del programa y las imágenes a procesar. La otra partición se usará como el sistema de archivos de este sistema operativo. Hay que tener en cuenta que antes de poder ejecutar cualquier programa que incluya alguna función de OpenCV, en el ámbito del trabajo todos tienen alguna, hay que ejecutar un comando para cargar las librerías en el Sistema Operativo. Este se tiene que ejecutar cada vez que se apague y encienda el sistema, ya que la configuración se añade al programa cargado en RAM y al apagar la placa esto se borra y se vuelve a cargar todo de SD. El comando es *export LD_LIBRARY_PATH=<ruta de la carpeta que incluye estas librerías>*.

Una vez hecho esto, se pasa a crear los ficheros con los que se dará vida al SO. Para ello, solo hay que seleccionar el sistema operativo que se desee instalar, en este caso Linux, en *System Configuration* dentro de la pestaña que sale a tal efecto al inicio del programa. En la figura se muestra cómo debe de estar configurado. También se puede configurar más adelante en el archivo .sdx. Aquí se debe seleccionar la opción *Generate SD image* para que se creen los ficheros necesarios para montar el sistema operativo.

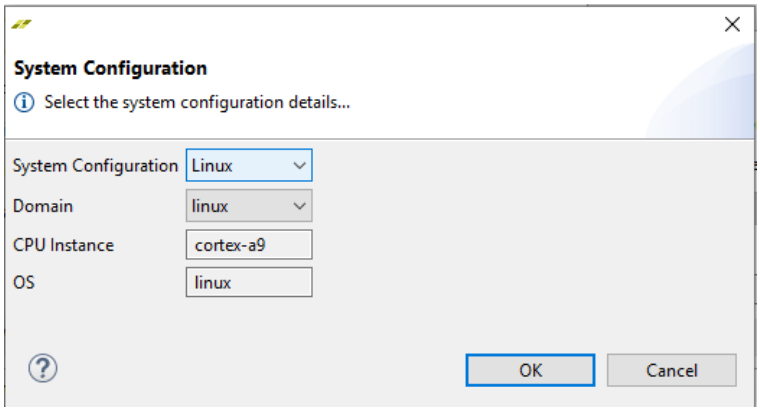


Figura 8:Configuración sistema operativo

Tras construir el proyecto sin errores, se abre el explorador de archivos y se selecciona la carpeta *<nombre_proyecto>/Release/sd_card*. En ella se encontrará todo los archivos necesarios para que el sistema operativo arranque y funcione sin problemas. Además se encuentra el ejecutable (.elf) del programa que se ha compilado en el proyecto.

En caso de estar activa la configuración de construcción *Debug*, habría que ir a la misma carpeta pero dentro de esta en vez de la de *Release*. Los ficheros se copiarán en la partición de la SD con formato fat32, se recomienda dar a esta partición al menos un GB para que luego el SO se pueda cargar de forma correcta. Tras finalizar la transferencia se extrae del ordenador y se introduce en la placa de desarrollo. Se configuran los *jumpers* de arranque para que la placa arranque desde la SD y se configura un puerto serie desde el ordenador que servirá como terminal para controlar todo el sistema operativo. Ya que este se usará como ventana de comandos para configurar y lanzar todos los programas. Los ficheros nombrados anteriormente sirven para:

- **BOOT.BIN**: en él se encuentra todo lo necesario para arrancar el sistema operativo y programar la FPGA. Por lo tanto si se van a lanzar dos programas, uno con particionado y otro con solo SW, hay que coger este archivo del proyecto que haya salido el particionado. Ya que si no, al intentar acceder a la parte HW para ejecutar una función acelerada, el programa devolverá un error de que no encuentra ese periférico. El solo SW puede funcionar sin problemas con la FPGA programada porque no hace uso de ella y a ese programa no le afecta nada que la FPGA este o no configurada para realizar una serie de operaciones.
- **Image.ub:** este incluye una imagen del Petalinux que se usará al arrancar la placa. Este se encuentra comprimido para ahorrar espacio aunque a la hora de cargarlo se expande para poderse usar. Este es solo una imagen del sistema y es siempre el mismo, sin importar si hay parte de FPGA programada o es todo SW.
- **<nombre_programa>.elf:** es el ejecutable que incluye el código del programa que se está desarrollando y se ha compilado en el proceso que ha dado como resultado estos archivos.
- **README.txt:** indica las instrucciones comentadas con anterioridad para arrancar el programa que se ha obtenido de su compilación.

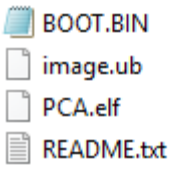


Figura 9: Archivos necesarios para la correcta instalación de SO en ZedBoard

6. Diseño Final

6.1 Algoritmo PCA

El algoritmo PCA (*Principal Components Analysis*) es un algoritmo usado en diversidad de aplicaciones como visión artificial o estadística. Este algoritmo es usado cuando se quiere procesar gran cantidad de datos en poco espacio de tiempo. En visión, por ejemplo, el tiempo de respuesta es crucial en muchos casos. Con este algoritmo se podrá conseguir este objetivo, ya que permite reducir la complejidad de cálculo de analizar todas las entradas al sistema porque reduce la dimensionalidad del problema quedándose solo con la información incorrelada, llamadas componentes principales en este algoritmo. Estas componentes principales se originan como una combinación lineal de las de partida y tienen la característica de ser independientes entre sí. Aunque se tiene que poder seguir representando la información del conjunto completo sin perder información, o que esta pérdida sea lo menor posible. Esta pérdida se cuantifica con la varianza de las componentes principales, a mayor valor más información de la original conservada en ellas.

La aplicación del algoritmo PCA en el campo de visión permite diferentes aplicaciones como puede ser la detección de nuevos objetos en una escena a partir de la extracción de las componentes principales de la imagen actual con las de un fondo de imágenes. Por tanto, es necesario construir un fondo de imágenes con el que se desea comparar y analizar una imagen. Así, a partir de un número de imágenes de fondo, se creará una matriz de transformación denominada U. Cuando llegue una imagen nueva (x), se usará esta matriz U para pasar la imagen a un nuevo espacio, con dimensionalidad la del número de columnas de U, y reconstruirla posteriormente con esta misma matriz. Luego se verá la diferencia entre la de entrada y esta imagen de salida. Con ello, se construirá un mapa de distancias que permitirá ver, una vez umbralizado y eliminado el ruido, dónde y cómo de grande es el objeto detectado, en caso de que lo haya. Si el porcentaje de área en la que se ha detectado un objeto es menor de un umbral, no hay objetos en la imagen. Se pone un umbral debido a que la varianza de las componentes seleccionadas no será la máxima, ya que no se coge todas las principales sino las más representativas. Esto se hace para ahorrar operaciones y poder hacer el algoritmo más rápido. Por lo que hay que asumir que habrá un cierto error siempre, aunque se introduzca una imagen con solo fondo, que en principio no debería tener error.

En vista de lo anteriormente expuesto, para poder aplicar este algoritmo, hacen falta un conjunto de imágenes previas para construir el fondo. Para ello, las imágenes tienen que ser captadas sin mover la posición de la cámara ni las condiciones luminosas de la escena. En el caso de que la luz en la imagen a analizar cambiase se debe hacer un proceso de actualización periódico de la imágenes que conforman el fondo de la escena sobre la que se está detectando objetos para poder ir adaptándose a las nuevas condiciones, asunto que se introduce cómo mejora del algoritmo tras tomar unos primeros resultados de tiempos de ejecución del algoritmo sin mejora, a costa de aumentar la carga computacional. Si el cambio de luz fuese muy brusco o la cámara se moviese, esta mejora no podría adaptarse porque se detectaría como si hubiese un objeto en la imagen, ya que hay mucha parte del fondo que ha cambiado, y esta mejora no actualiza fondo si siempre detecta objeto en la imagen. Habría que arrancar de nuevo el algoritmo para que se adaptase al nuevo fondo.

En base a lo anteriormente expuesto, se pueden diferenciar y por tanto clasificar, dos partes en este algoritmo. La primera se llamará **offline** y en ella se calculará la matriz de transformación a partir de un cierto número de imágenes de fondo. La segunda, llamada **online**, consiste en proyectar en el espacio transformado la imagen que se recibe (x), usando la matriz de transformación (U) calculada en la parte offline, y volverla a recuperar(x_r) a partir de esa proyección. A continuación, se calculará el error de reconstrucción de cada píxel ($\varepsilon_j = |píxel_j - píxel_{jr}|$). Con esto, se creará un mapa de distancias (MD) que me permita ver dónde está el objeto que hay en la imagen. El nuevo objeto se encontrará donde los errores de reconstrucción de cada píxel sean mayores. Ya que el valor del píxel de la imagen procesada y el de la imagen de fondo su valor diferirá mucho y el proceso de proyección y recuperación harán que esta diferencia se magnifique más. Para poder verlo mejor, se le aplicará a este MD un umbral. Además, a la imagen binarizada se le aplicará un proceso de erosión y dilatación para eliminar parte del ruido impulsivo que pueda haber.

Tras esta pequeña presentación de lo que se va a realizar en cada una de las partes, se procede a explicar detalladamente cómo se implementa la funcionalidad de cada parte.

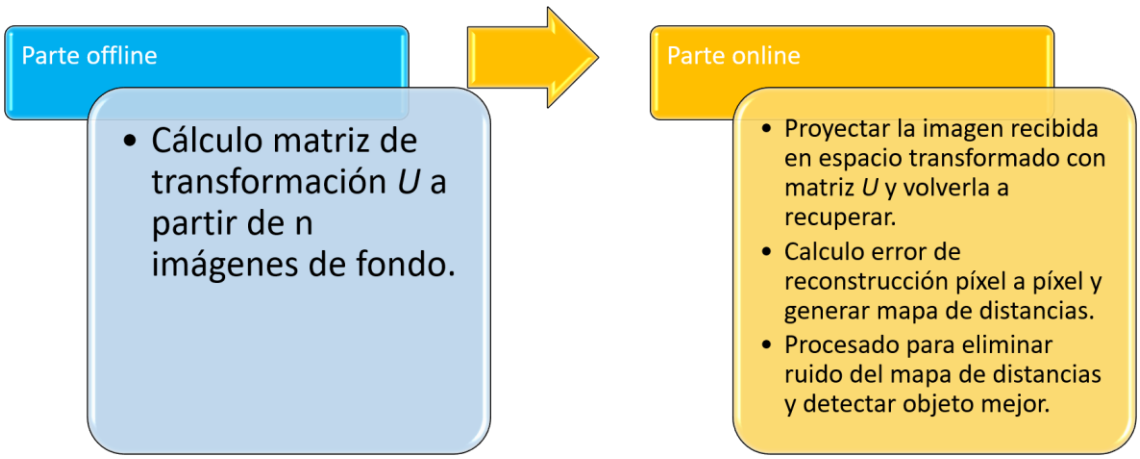


Figura 10: Resumen de los pasos más importantes a realizar en cada parte del algoritmo PCA

6.1.1 Parte offline[6]

El punto clave de este algoritmo es encontrar una matriz de transformación para llevar a cabo el paso al espacio transformado, con menor dimensionalidad, y poder recuperar la imagen desde este. Esta matriz se crea a partir de varias imágenes de entrada en la que no hay objetos, lo que se denomina imagen de fondo. En este caso se ha decidido tomar 8 imágenes para formar la matriz de transformación. Se ha decidido tomar un número potencia de 2 porque a la hora de hacer la media de las 8 imágenes la división es una simple operación de desplazamiento. Cosa que reduce mucho la complejidad de cálculo. En este algoritmo se trabaja con las imágenes como matrices de dimensiones SIZEx1. Siendo SIZE el número de píxeles de la imagen de entrada. Por lo que el primer

[6] Todos los pasos que se describen en las dos partes se han extraído de [Bravo,2007]

paso será pasar de la matriz rectangular que contiene la información de la imagen a este vector columna para poder trabajar correctamente con las imágenes.

Cada imagen de fondo será una columna de una matriz de datos de entrada (I). Por lo que está será de tamaño SIZEx8. Ahora se calcula la media (ψ) de cada uno de los píxeles, en la matriz es la media de los elementos de cada una de las filas. Está se le resta a cada elemento de la matriz de entrada por filas. O sea, la media de la fila N se le resta a cada uno de los elementos de la fila N. Esto se hace porque los datos de entrada a este algoritmo deben tener la media nula. La matriz resultante se llamará A.

Para generar la matriz U, se tienen que calcular los autovectores de la matriz de covarianza de A. Siendo la matriz de covarianza: $C = \frac{1}{M} * A * A^T$. En este punto surge un problema, ya que, si se hiciese así, la matriz obtenida sería de tamaño SIZExSIZE. Este tamaño supone un gran cálculo computacional por ser una matriz con un elevado número de elementos. Por lo que se recurrirá a la propiedad de que los autovalores de $(A * A^T)$ son los mismos que los de $(A^T * A)$. Primero se computan los de $(A^T * A)$, con lo que se obtendría la matriz V. Esta es de tamaño 8x8. Luego, para poder obtener la matriz U deseada, hay que multiplicar por la matriz A. Por lo que $U = A * V$. Esta nueva matriz ya sí tendrá las dimensiones deseadas (SIZEx8) para la matriz de transformación.

No hace falta conservar todas las columnas de U, ya que de esta forma, se tendría toda la información incorrelada de la imagen original, cosa que no es necesario. Ya que el porcentaje de acierto puede ser bueno asumiendo una cierta pérdida que permitirá reducir el coste computacional en cierta medida. Por lo que se opta por reducir el número de componentes principales a usar. Para la elección de cuántas componentes son necesarias para que esta pérdida no sea significativa, se usará el método RMSE (raíz del error cuadrático medio). Este se calcula como la división de la suma de los N autovalores más significativos entre la suma total de los autovalores. El valor mínimo de RMSE que se ha considerado oportuno para que no se dispare el error por no conservar la suficiente información es del 95%. En este caso, este valor se supera al incluir 4 componentes. Ya que si se toman menos componentes, se obtiene un valor de RMSE menor, lo que implica mayor error de reconstrucción. Esto no interesa, ya que complica la detección de objetos por no tener el error causado por no coger todas las componentes principales en valores pequeños. Si se cogiesen más componentes el error sería menor pero aumenta la complejidad, ya que hay una dimensión más en el espacio transformado, y la mejora no es tan importante. Por lo que la matriz de transformación usada en este algoritmo será de tamaño SIZE x 4. Esta estará formada por las columnas que contienen los autovectores que están asociados a los 4 autovalores de mayor valor. Todas las operaciones que hay que realizar en esta parte quedan resumidas en la figura siguiente.

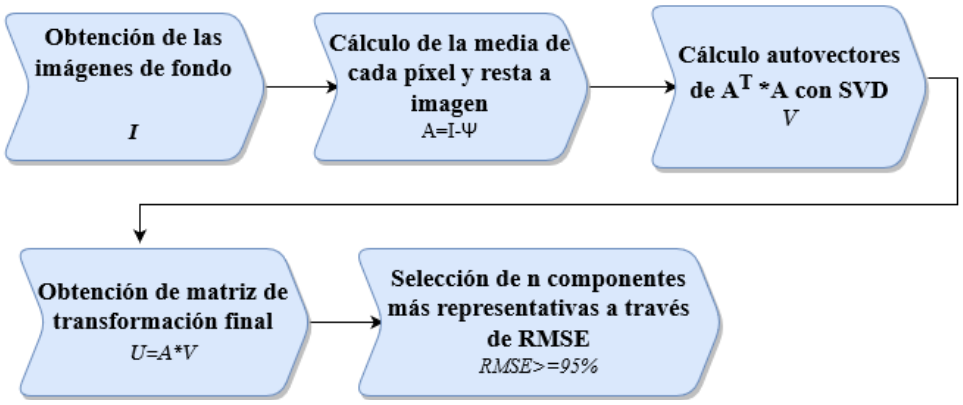


Figura 11: Diagrama creación matriz de transformación U, parte offline

6.1.2 Parte online

Una vez se tiene la matriz de trasformación totalmente construida, se pasa a la parte online del algoritmo. En ella, se va procesando las imágenes que llegan al sistema y se va viendo si hay objetos. Cada cierto tiempo, si la mejora está incluida, se irá actualizando el fondo si la imagen que se elige candidata no tiene objetos. En caso de tenerlos, no se actualiza el fondo y se continúa el procesado de imágenes de forma normal. Hay que tener en cuenta que si la imagen es apta para incluirla en el fondo (no tiene objetos), se lanza una nueva parte offline para recalcular la matriz U desde cero con los nuevos datos de fondo. Lo que supondrá que el valor de la media de cada píxel cambia. Antes de recalcular la matriz de transformación, hay que sumar a la matriz A la media que se tenga para cada fila. Ya que este cálculo se realiza como si se hubiera cambiado toda la matriz y lo que se tiene son 8 imágenes nuevas con los valores extraídos directamente de ellas. Si no se hace esto, el cálculo obtenido será erróneo y provocará que el proceso de detección se corrompa por no tener una matriz de transformación válida.

A la imagen que llega se le resta el valor de la media de cada variable del fondo que se tenga en ese momento ($\varphi = x - \psi$). Ya que la proyección de la imagen se tiene que hacer en las mismas condiciones con las que se ha procesado las imágenes del fondo actual. De esta forma, si es fondo, el error cometido será pequeño, aunque no nulo porque no se está usando en la trasformación todas las componentes obtenidas, sino las más representativas. Esta proyección se hace de la siguiente manera: $\Phi = U^T * \varphi$. Una vez realizado, esta proyección, Φ tendrá un tamaño de 4x1, ya que se han concluido que, de la matriz total U, se considerarán las 4 componentes más significativas. Con ellas, son con las que se opera. A partir de esta se recupera la imagen haciendo $\varphi' = U * \Phi$. A la resultante, se le suma la media para poder volver a tener una imagen como la de entrada al sistema (x_r) y así comparar el error de reconstrucción fácilmente.

Ahora, para ver si hay objeto o no en la imagen y dónde está ubicado, se calcula el mapa de distancias (MD). Este se obtiene como la diferencia entre el valor del píxel original (x_j) y el del reconstruido (x_{rj}) . Esto queda ilustrado en la fórmula $MD[j] = |x_j - x_{rj}|$. Por lo que *MD* es un array unidimensional que contiene SIZE elementos, uno por cada píxel de la imagen. Al tener estos valores para todos los píxeles de la imagen, se le aplica un umbral para ver si el error de reconstrucción se descarta, ya que es un error

originado por no haber tomado todas las componentes, o se tiene en cuenta por ser un error de reconstrucción alto, lo que indicará que hay objeto detectado en ese píxel. En este caso se optó por poner un valor estático de 60 a este umbral. Ya que se observó que si se ponía un valor más pequeño se clasificaban píxeles con errores producidos por no tener todas las componentes como píxeles con objeto, por lo que la detección de objetos era mala y no se podía hacer de forma correcta. Si se ponía más alto, se dejaban de detectar píxeles con objeto como tales, por lo que se perdía parte del objeto. Se deja como trabajo futuro que este umbral sea dinámico para conseguir que la clasificación sea lo mejor posible y se pueda separar lo mejor posible el error interno del producido por detectar objeto. Tras ello, se somete a la imagen binarizada a un proceso de erosión y posterior dilatación para eliminar la mayor parte del ruido que pudiese quedar. En este caso, la máscara tanto para la erosión como para la dilatación tiene esta forma: $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$. Donde sus dimensiones son de 3x3 y todos los elementos valen 1. Esto quiere decir que se tendrán en cuenta todos los valores que rodean al píxel que se está computando para elegir su valor. Además todos tienen el mismo peso. Solo se hace una iteración en cada uno de los dos procesos. Aplicar estas operaciones morfológicas es muy interesante ya que se permite eliminar completamente el ruido *shot* que pueda haber surgido en la clasificación y poder así poder tener solo los píxeles que corresponden con el objeto. Lo que hace su detección y seguimiento mucho más sencilla Para ilustrarlo se incluyen unas imágenes de antes y después de este procesado.

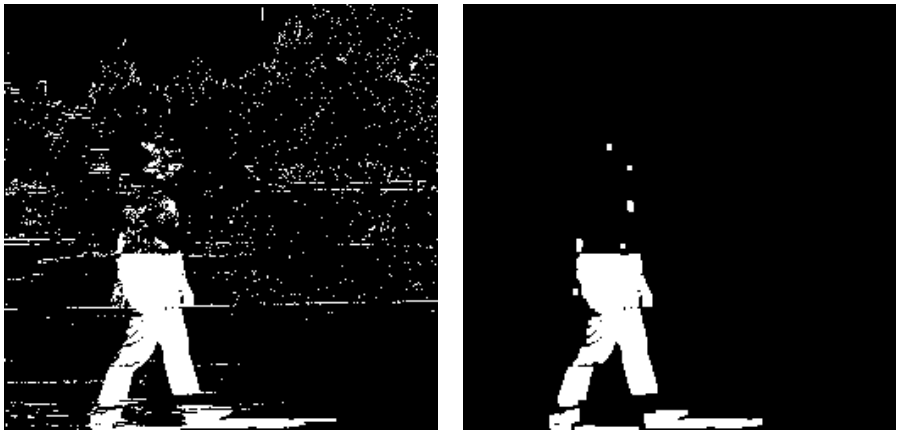


Figura 12: Mapa de distancias antes (izquierda) y después (derecha) de aplicar las operaciones morfológicas

En esta imagen procesada, se contará el número de píxeles en los que se tiene objeto después de este procesado (blancos se le llama en el código). Este valor se dividirá por el número de píxeles de la imagen (SIZE) y se multiplicará por 100 para obtener un porcentaje. Si el área en la que se ha detectado objeto supera un umbral, 1,15% en este caso, se considerará que hay objeto en la imagen. En caso de ser menor, lo único detectado se considerará ruido interno del proceso y se decidirá que no hay ningún objeto en la imagen. Se ha optado por poner este valor ya que, a pesar de eliminar el ruido con las operaciones morfológicas, no se elimina todo y a veces aparecen pequeñas áreas blancas que son ruido. Tras hacer una observación de qué porcentaje del total de la imagen representaban este ruido no eliminado, se determinó que no representaba más de 1%, en

la mayoría de los casos. Por lo que se impuso un margen que hiciese que los falsos positivos por acumulación de ruido se redujeran al mínimo. Tras esto, si la imagen no es candidata para actualizar el fondo, se pasa a procesar la siguiente imagen. Empezando otra vez todo el proceso descrito en este apartado. Esto se hará hasta que se acaben las imágenes del set o se decida parar la ejecución del algoritmo.

Al hacer multiplicaciones de matrices en las que el número de elementos es elevado, los valores obtenidos son bastante grandes ya que es una suma de muchas multiplicaciones de valores, en valor absoluto, mayores que 1. Esto se ve sobre todo a la hora de proyectar en el espacio transformado. Por lo que, tras proyectar y tras recuperar la imagen del espacio transformado, se somete a esta a un proceso de normalización. Gracias al cual los valores de cada elemento oscilan entre -255 y 255, rango de valores de la imagen de entrada menos la media del fondo. Aplicando la siguiente ecuación: $\varphi'[j] = round(\varphi[j] * \frac{255}{\text{máx}(|\varphi|)})$. De esta forma no aumentan en gran medida los valores al volver a recuperar la imagen. Además el resultado se redondea al entero más próximo para que los valores sean del mismo tipo que los valores de entrada pero el error cometido sea el menor posible, ya que este redondeo también introduce cierto error.

A modo resumen, se incluye un diagrama con las operaciones que se van a realizar en la parte online del proceso.

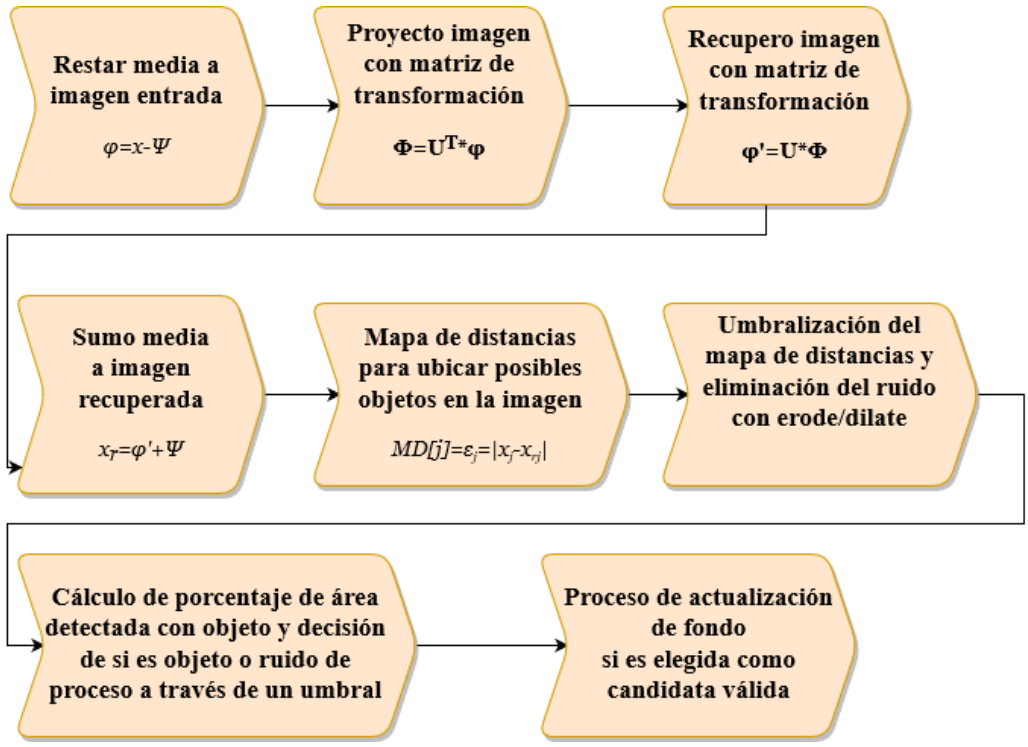


Figura 13: Diagrama de comportamiento de parte Online de PCA.

Para ver cómo de bueno este algoritmo detectando objetos, se introduce unas variables que permitirán llevar un registro de los falsos positivos (algoritmo detecta objeto pero no lo hay) y de los falsos negativos (algoritmo no detecta objeto cuando sí lo hay). Estas aumentan su valor si lo que decide el algoritmo no coincide con lo que la clasificación manual de las imágenes dice. O sea, si la clasificación manual dice que hay

un objeto en la imagen, pero el algoritmo decide que no lo hay, se está ante un falso negativo. Por lo que habría que sumar uno a esta variable. Los falsos positivos se producen cuando el algoritmo decide que hay objeto pero la clasificación manual determina que no lo hay. La suma de estos dos da como resultado el número de errores de detección que se han producido en la ejecución del algoritmo. Para poder verlo con más claridad se sacará la tasa de acierto. Esta es $\%_{acierto} = 100 * (1 - \frac{N_{errores}}{N_{imagenes\ procesadas}})$. Cuanto más se acerque al 100% mejor será. La forma de ver si la detección es errónea o no es comparando lo devuelto por el algoritmo con una clasificación manual realizada observando imagen a imagen si hay objeto o no.

6.2 Procedimiento de trabajo

Lo primero que hay que hacer es desarrollar un código que implemente el algoritmo que se quiere particionar en HW/SW, en este caso será el algoritmo PCA. Para ello, se usará un código escrito en lenguaje C++ y funciones de OpenCV. A pesar de no tener, aparentemente, ninguna restricción sobre los tipos de datos y funciones que se pueden usar, no hay que olvidar en ningún momento que lo que se está escribiendo puede que interese implementarlo en hardware. Por ello, hay que evitar usar elementos de programación que en una FPGA, a priori, no es posible o es complicado de implementar, como puede ser punteros o asignación dinámica de memoria. Los bucles y los arrays usados deben tener un tamaño conocido en tiempo de compilación ya que sino su implementación no es posible (no se pueden asignar recursos de la FPGA dinámicamente). Debido a que no se sabe cómo están implementadas las funciones de OpenCV y se presupone que, algunas de ellas, pueden hacen uso de la memoria dinámica, se intentarán usar lo menos posible. En este trabajo solo se emplean para leer la imagen y realizar las erosiones y dilataciones usadas para eliminar el ruido del mapa de distancias. El resto está programado con arrays y bucles de tamaños fijos, para así conocer en tiempo de compilación su tamaño y poder tener acceso a todo el código. Así se podrá incluir sentencias pragmas donde se considere más oportuno sin problemas y tener más control de lo que hace cada operación.

Se comenzará escribiendo el programa en MATLAB, donde se verificará que el algoritmo implementado funciona como se espera. Además se comparará el resultado de calcular los autovalores y lo autovectores para la matriz de transformación U con la función *svd* (la que luego se aplicará en el código en C++) y *pca* (función de MATLAB creada para calcular está matriz desde los datos de entrada directamente). Los resultados obtenidos tras probar las dos funciones son muy parecidos ya que el error máximo de los autovectores asociados a los autovalores más representativos no supera el 5%, aunque en la mayor parte de los valores está por debajo del 2,5% de diferencia con respecto al valor devuelto por la función *pca*. Por lo que a partir de ahora se usará la función *svd* (en el código en C++ llamada dsvd [Buja,1997]) dado que ésta es más comúnmente usada y se puede encontrar con más facilidad para poder implementarla directamente. Además se muestra por pantalla una ventana con el error total de reconstrucción de la imagen y otra donde se selecciona un pixel de la imagen por el que vaya a pasar el objeto para ver cómo el error de reconstrucción de éste va cambiando a lo largo del set de imágenes. Lo más

interesante de esta segunda pantalla es ver que, cuando el píxel forme parte del objeto, el error de reconstrucción aumenta en gran medida. En las siguientes etapas del desarrollo del algoritmo, no se usará este error total para detectar si hay objeto o no en la imagen sino que se usará el explicado con anterioridad de ver cuanta área de toda la imagen se detecta con objeto. Esto es debido a que este error fluctúa mucho y muy rápidamente. Además, su valor no aporta mucha información de cómo es el objeto, u objetos, detectados de grande y no tiene un filtro para eliminar ruido que pueda reducir al mínimo los errores de decisión al mínimo. Se decide usar la técnica de ver el área de la imagen en la que se ha detectado objeto, ya que esta da información de lo grande que puede ser el objeto u objetos que hay en la imagen y se ha eliminado la gran mayoría del ruido. Esto dará valores con variaciones más suaves y más acordes a lo que está pasando en realidad con el objeto. En la siguiente figura se muestra una captura de lo que se ve al ejecutar el programa de MATLAB.

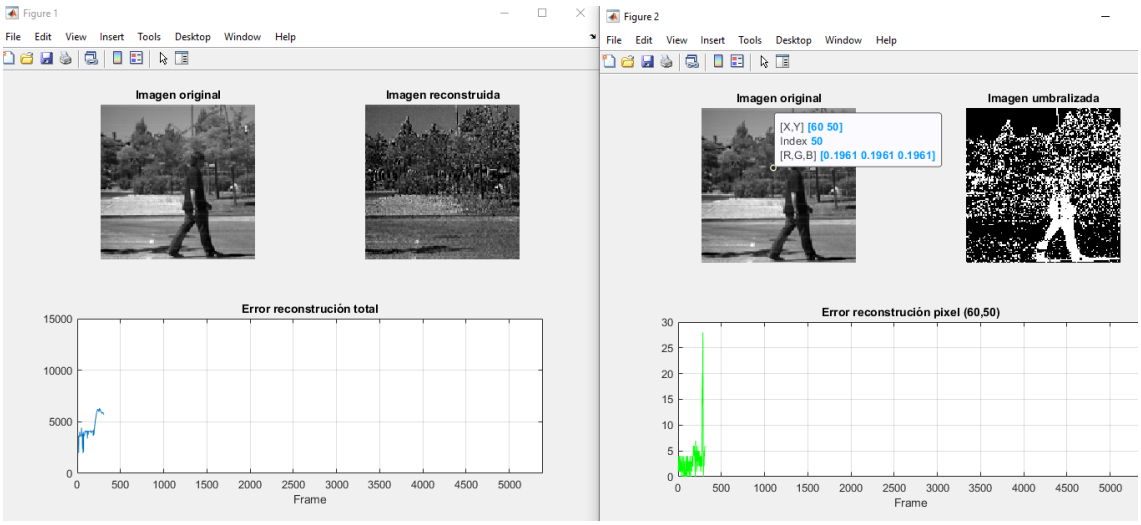


Figura 14: Ejecución Matlab. Error total e imagen reconstruida (izquierda). Mapa de distancias y error en un píxel (derecha)

Tras comprobar que el programa creado era válido, se pasó a escribir en C++ el algoritmo. De esta forma, luego se puede importar de forma sencilla a SDSoC. Para su desarrollo se optó por realizarlo en Eclipse bajo una máquina virtual de Linux. Así, se podía ver de forma sencilla si las funciones y los procedimientos que se estaban siguiendo se podían implementar de forma exitosa luego en la descarga en placa ya que el sistema operativo usado, aunque mucho más simplificado el del SoC, es el mismo. Las imágenes venían rotadas 90º en sentido antihorario. En el caso de MATLAB se eligió ponerlas en su posición correcta, pero, para no sobrecargar el algoritmo y usar el menor número de funciones de OpenCV, aquí se dejan rotadas, al igual que estarán para la entrada del sistema implementado en el SoC.

Tomando de base la matriz de autovectores devuelta a por la función *pca* de MATLAB, se implementa un pequeño algoritmo para igualar los signos de cada una de las columnas de la matriz devuelta por la función *pca* y la de *dsvd*. Esto es debido a que algunos de los autovectores devueltos por cada una de estas dos funciones tienen signos opuestos. Una vez más, los signos que se tomarán como buenos son los devueltos por la función *pca*. Así se consigue que el error de cálculo del algoritmo que se está implementando sea lo menor posible. Se considera que el procedimiento bueno para calcular la matriz U sin error es el devuelto por la función *pca* de MATLAB. Es verdad

que hay pequeñas variaciones en los valores de estas dos funciones, pero como son pequeños, se consideran errores asumibles de precisión de datos. Otra cosa que se debe de hacer antes de proceder con la creación de la matriz U, es ordenar los autovalores y los autovectores de mayor a menor valor. Se recuerda que a la función *dsvd* se le pasa una matriz más pequeña para acelerar el cálculo. Esta luego se multiplicará por la matriz de las muestras tomadas como fondo, a las que ya se le ha restado la media de cada píxel, y se obtendrá U. Todo lo demás es igual que el procedimiento implementado en MATLAB solo que haciendo la conversión de las funciones de MATLAB a su equivalente en C++ o de OpenCV. La salida se obtiene en Eclipse es la mostrada en la Figura 15.

En el mapa de distancias solo se ve la mitad de la persona debido a que el nivel de gris de la persona al pasar la imagen a escala de grises y el del fondo son similares. Por lo que la diferencia de nivel de gris entre el fondo y el objeto no son muy grandes por lo que al hacer la transformación y recuperarla el error no es muy elevado y no se detecta objeto ahí. Donde más se puede observar es en las piernas de esta, ya que el cambio de nivel de gris en este caso si es más acusado. En la imagen derecha se puede observar el mapa de distancias procesado para eliminar el ruido casi al completo y solo ver el objeto detectado de forma clara. Debajo de las tres imágenes, se puede observar la salida del algoritmo en la que se indica el porcentaje de área en la que hay un objeto distinto al fondo (parte blanca de la imagen llamada proc) y, entre paréntesis, la decisión que ha tomado el algoritmo respecto a si hay o no objeto en el frame que analiza.

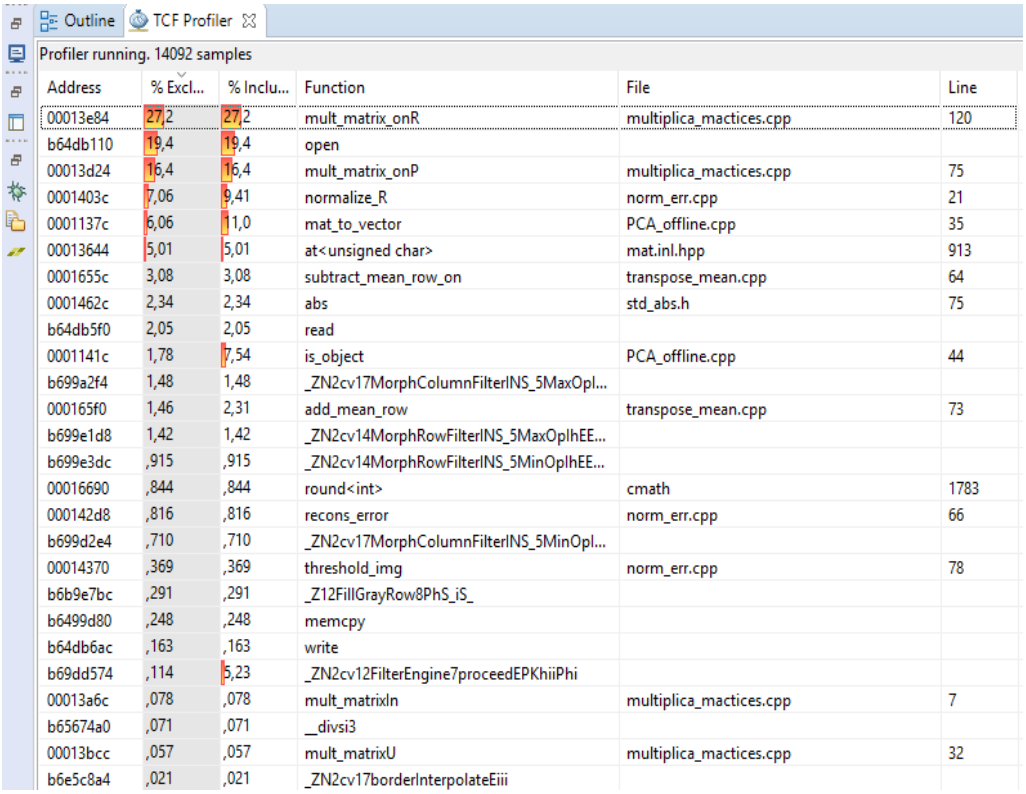
Figura 15: Salida eclipse.

Al comprobar que el algoritmo implementado en C++ funciona según lo esperado, el siguiente paso es proceder al uso de SDSoC. Para ello, se creará un proyecto en SDSoC al que se importarán todos los archivos C++ creados juntos con las cabeceras creadas a tal efecto de la forma que se ha explicado anteriormente en *Primeros pasos con SDSoC*. Antes de comenzar el estudio para ver qué llevar HW y qué a SW, se procede a la construcción del proyecto para ver que todo está bien configurado y genera un programa válido. Este, si se desea, se puede copiar en la SD como se enuncia anteriormente y hacer una prueba rápida de que todo funciona correctamente.

Una vez asegurado el correcto funcionamiento se va a empezar el estudio de ver qué es más interesante pasar a HW y qué dejar en SW (fase de particionado del diseño). Por ello, se arranca el *Debugger* y se inicializa la herramienta de perfilado *TCF Profiler*. Cuando está todo configurado se lanza la ejecución del programa y se deja que procese

todas las imágenes del dataset. Esto puede llevar un par de minutos porque son muchas imágenes y se está usando el modo *Debug*, que hace que el programa se ejecute más lentamente. Además, todo se está procesando en SW y por tanto no está optimizado. Se recomienda poner un *breakpoint* en el *return* del "main" para que el programa se detenga y se pueda ver sin problemas los resultados devueltos por el *TCF Profiler*. Si no se hace así, el programa concluye y el perfilador se cierra automáticamente y no se pueden observar los datos extraídos de este.

Concluida la ejecución del programa, se observa qué funciones han estado más tiempo procesándose. Para ello se analiza la columna *%Exclusive* de esta pestaña la cual indica el tiempo que se ha empleado en ejecutar realmente código propio de esta función y no contando también el de las posibles funciones que llama en su interior. En el caso de esta ejecución, como se puede observar en la Figura 16, las funciones que más han tardado en ejecutarse son: *mult_matrix_onR*, *mult_matrix_onP* y *normalize_R*. Por tanto, estás serán las candidatas a ser trasladas a HW ya que cualquier mejora en ellas tendrá mayor repercusión en la mejora de la aceleración global que si se mueven a HW otras que su tiempo de proceso es menos significativo. Solo se toman tres para pasar a HW y no hasta completar los recursos de la FPGA, ya que se optó por pasar a HW las funciones que más tiempo de CPU consumieran hasta llegar a sumar más del 50% de la ejecución del programa. En concreto, suman el 50,66% del total. La función *open*, que es la segunda que más tiempo de CPU consume, no se ha cogido ya que no es una función creada por el desarrollador sino que es una del SO, como se puede observar en esta, no pone en que archivo se puede encontrar.



Outline | TCF Profiler

Profiler running. 14092 samples

Address	% Excl...	% Inclu...	Function	File	Line
00013e84	27,2	27,2	mult_matrix_onR	multiplica_mactices.cpp	120
b64db110	19,4	19,4	open		
00013d24	16,4	16,4	mult_matrix_onP	multiplica_mactices.cpp	75
0001403c	7,06	9,41	normalize_R	norm_err.cpp	21
0001137c	6,06	11,0	mat_to_vector	PCA_offline.cpp	35
00013644	5,01	5,01	at<unsigned char>	mat.inl.hpp	913
0001655c	3,08	3,08	subtract_mean_row_on	transpose_mean.cpp	64
0001462c	2,34	2,34	abs	std_abs.h	75
b64db5f0	2,05	2,05	read		
0001141c	1,78	7,54	is_object	PCA_offline.cpp	44
b699a2f4	1,48	1,48	_ZN2cv17MorphColumnFilterINS_5MaxOpI...		
000165f0	1,46	2,31	add_mean_row	transpose_mean.cpp	73
b699e1d8	1,42	1,42	_ZN2cv14MorphRowFilterINS_5MaxOpIhEE...		
b699e3dc	,915	,915	_ZN2cv14MorphRowFilterINS_5MinOpIhEE...		
00016690	,844	,844	round<int>	cmath	1783
000142d8	,816	,816	recons_error	norm_err.cpp	66
b699d2e4	,710	,710	_ZN2cv17MorphColumnFilterINS_5MinOpI...		
00014370	,369	,369	threshold_img	norm_err.cpp	78
b6b9e7bc	,291	,291	_Z12FillGrayRow8PhS_iS_		
b6499d80	,248	,248	memcpy		
b64db6ac	,163	,163	write		
b69dd574	,114	5,23	_ZN2cv12FilterEngine7proceedEPKhiiPhi		
00013a6c	,078	,078	mult_matrixIn	multiplica_mactices.cpp	7
b65674a0	,071	,071	__divsi3		
00013bcc	,057	,057	mult_matrixU	multiplica_mactices.cpp	32
b6e5c8a4	,021	,021	_ZN2cv17borderInterpolateEiii		

Figura 16: Resultados obtenidos del TCF profiler tras una ejecución completa del programa.

Es verdad que con estas no se usarán todos los recursos de la FPGA, pero las demás funciones, de forma individual, suponen un porcentaje bastante pequeño del total de ejecución. Por ello, la labor de pasarlas a HW no se considerará interesante, ya que la mejora obtenida no va a poder ser suficiente significativa y con tres funciones se puede ilustrar de forma bastante buena la forma de trabajar con este programa. Tras haber elegido las funciones a pasar a HW se puede pasar al siguiente paso: optimizar su implementación en HW. Para conseguir la mayor aceleración posible con el menor uso de recursos HW.

6.3 Elección de parámetros

En este apartado se va a explicar los parámetros elegidos para el diseño. El primer valor que hay que elegir es el número de imágenes que se van a tomar como fondo de la imagen (N_IMG_FONDO). Debido a que se quiere tener un número potencia de 2 para que las divisiones, en caso de que se hagan en hardware, sean simples desplazamientos a la hora de calcular la media. Para no sobrecargar el algoritmo con muchos datos para calcular la matriz U, se ha tomado la decisión de que el fondo este formado por 8 imágenes diferentes. O sea que la matriz U será, como muy grande, de SIZEx8.

El tamaño de la imagen (TAM_IMG) elegido es de 256x256 y un solo canal (escala de grises) de 8 bits, ya que las imágenes que son usadas como entrada tienen esta resolución y número de canales. Se ha optado por elegir procesar las imágenes en escala de grises porque, aparte de que estas ya venían en un solo canal, procesar imágenes en color implicaría hacer un procesado paralelo en los tres canales, uno por color, lo que complicaría la implementación y, en la finalidad de este trabajo, no aportaría ningún valor añadido. En cuanto al tamaño de la imagen, no se selecciona otro más pequeño ya que esta resolución se puede considerar una situación real en la que las imágenes son captadas con una cámara de baja resolución. Tener imágenes de entrada con menos resolución sería algo que ayudaría a reducir en gran medida el consumo de recursos, ya que los datos con los que se trabaja consumen gran cantidad de espacio, pero sería imágenes muy pequeñas en las que no se podría ver bien los objetos detectados y estarían poco definidos. La justificación de elegir la forma de la imagen cuadrada es porque es más sencillo trabajar con matrices cuadradas que si no lo son, ya que al recorrer una matriz habría que tener muy claro la forma de recorrer la matriz para poner los límites de cada uno de los bucles de forma correcta y no salirnos de la zona de memoria de la matriz por poner mal los límites. Esto provocaría obtener resultados erróneos sin que, aparentemente, el proceso de cálculo esté mal.

El número de componentes representativas (COMP) será de 4 ya que, como se comentó en el apartado de parte offline de PCA, con esta cantidad de elementos representativos el RMSE es del 96%. Por lo que se tendrá suficiente información para poder hacer el proceso de proyección y reconstrucción del espacio transformado sin cometer mucho error. Se elige el menor número de componentes que cumple con la probabilidad mínima, para poder ahorrar el máximo número de recursos y tiempo de ejecución.

El *while*, dentro del que se implementa la funcionalidad de la parte online del algoritmo, concluye cuando llega a las 5390 ya que es el número de imágenes (N_IMG_SET) que hay en el set elegido para probar el funcionamiento de este algoritmo. El valor del *threshold* (THRES) se elige de forma empírica. Probando varios valores, se llegó a la conclusión de que 60 para la umbralización del mapa de distancias eliminaba bastante del ruido producido por el proceso (redondeo en procesos de normalización, no coger todas las componentes principales) y permitía ver con claridad los objetos detectados. Tras justificar el valor de todos los parámetros importantes del algoritmo, se incluye una tabla en la que se resumen todos ellos para mayor claridad.

Parámetro	Definición	Valor
N_IMG_FONDO	Número de imágenes que conforman el fondo contra el que se van a comparar las imágenes	8
TAM_IMG	Tamaño de la imagen, en alto y ancho.	256x256
COMP	Número de componentes principales consideradas representativas	4
N_IMG_SET	Número de imágenes del set que se está procesando	5390
THRES	Valor del *threshold* usado para umbralizar el mapa de distancias	60

Tabla 1:Valores de parámetros principales del sistema

Se ha optado por usar un SO sobre el que correrá el algoritmo que se está implementando. Esta decisión se ha tomado ya que el algoritmo necesita unas imágenes de entrada a procesar, a modo de simular que está procesando lo que está grabando una cámara. Gracias a que se usa un SO, el sistema de archivos ya viene montado, así que, a la hora de trabajar con un fichero de imagen, se pueden seguir usando las mismas funciones que cuando se han hecho las pruebas de funcionamiento del algoritmo con el ordenador en Eclipse, en una máquina virtual de Linux, y similares a las usadas en MATLAB. Sin sistema operativo habría que montarse un sistema de archivos propio o pasar las imágenes a memoria a través de algún procedimiento, como puede ser la opción *"program flash"* de SDSoC e ir leyendo trozos de memoria del tamaño de la imagen. Ya que con este programa la implementación de un sistema operativo es bastante sencilla se opta por esta opción. Es verdad que conllevará una pequeña sobrecarga del sistema pero no supone ningún problema, ya que la opción de un sistema *standalone* llevaría muchísimo más tiempo implementarla (se deja como trabajo futuro) y no hay una restricción temporal establecida a cumplir.

Para ver cómo de bueno es el particionado que se está realizando se van a definir una serie de KPI. El primero que se tendrá en cuenta es el speed up conseguido ya que a mayor valor, mejor será el particionado ya que se habrá conseguido reducir al máximo el tiempo de procesado del algoritmo. Otro que también se tendrá en cuenta es la relación recursos-latencia. Es verdad que se intentará priorizar que la latencia sea menor. Pero no es buen diseño aquel que para reducir la latencia en unos pocos ciclos necesita una cantidad mucho más elevada de recursos. Por lo que se intentará conseguir latencia lo

más baja posible pero, para llegar a este objetivo, no haya que hacer un consumo de recursos descompensado con la reducción de tiempo. El último factor que se tendrá en cuenta es la tasa de acierto del algoritmo detectando objetos en imagen. Este se medirá con la tasa de acierto, que se intentará que no sea inferior al 95% para la ejecución de todo el set de imágenes. En este caso se considerará una clasificación buena por parte del algoritmo implementado.

6.4 Aritmética usada

Este apartado se explicará los tipos de datos elegidos para cada una de las variables que se usan en el desarrollo de la aplicación y por qué se ha decidido cada tipo para cada variable. En SDSoC, se permite trabajar, tanto para las funciones destinadas HW como en las destinadas a SW, en coma fija, enteros o como flotante. Debido a la mayor precisión de un número en coma flotante se intentará usar este tipo de datos cuando se necesite un número decimal. En caso de ser demasiado el consumo de recursos y no poderse asumir para conseguir tener en HW todas las funciones que se quieren implementar, se pasaría todo a coma fija. Pero en este caso se ha conseguido implementar todo con coma flotante sin problema de uso elevado de recursos. En concreto se han usado *float*, ya que se necesitaba precisión pero el valor de los datos que se van a introducir en estas variables no desbordaba su rango de valores. En cuanto a los enteros, se usará aquel que primero cumpla que los datos que tiene que almacenar no se salen del rango que se puede representar con este. Como a continuación se va a empezar a hablar de los distintos ficheros en los que se encuentra el código que implementa el algoritmo desarrollado, se incluye en este punto un esquema que muestra la jerarquía de estos para poder ubicarse en todo momento. Esta es bastante simple. Está el archivo "PCA_offline.cpp" y de él dependen todos los demás, ya que contienen el cuerpo de las funciones, en el caso de los .cpp, que se usan en el programa principal y los *defines* e *includes* necesarios para parametrizar todo el algoritmo y poder usar todas las funciones que el algoritmo necesita, en el caso del .hpp.

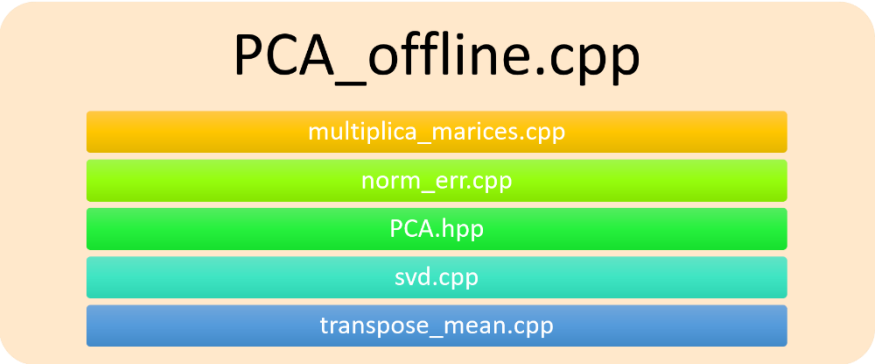


Figura 17: Jerarquía de archivos del proyecto

Las primeras variables que se pueden observar en el programa principal, que se puede encontrar en el capítulo 8, son variables usadas durante todo el programa para control de posición, errores o datos los datos que entran y salen del programa. La primera

es ‘pos’ usada para ver el frame que está procesando el programa en cada momento. Este no puede ser negativo y como mucho valdrá 5390, imágenes que contiene el set, así que el tipo de dato que mejor se ajusta es *unsigned short*. Las variables ‘file’ y ‘objeto’ se usarán para guardar cadenas de caracteres ASCII y ver si hay objeto o no en el frame n-ésimo respectivamente. Por lo que el tipo de dato más ajustado es *char*, ya que los caracteres ASCII ocupan 8 bits y los valores de ‘objeto’ son o 1 para decir que hay objeto o 0 si no lo hay. La variable ‘col’ indica la columna que lleva más tiempo sin modificarse de la matriz ‘A’, por lo que cuando se vaya a realizar una modificación de la matriz, se observará esta variable para ver que columna modificar. Como solo tiene 8 columnas, con asignarle el tipo de dato *char* es suficiente para que no desborde. Por último esta ‘frame’ que es de tipo *cv::Mat*, que se usará para trabajar con las funciones de lectura y escritura en archivos de OpenCV. En la siguiente tabla queda resumido todo lo expuesto en este párrafo.

Variables control programa		
Variable	**Descripción**	**Tipo dato**
pos	Indica el frame que está procesando el algoritmo en cada momento	*Unisgned short*
file	Ruta del archivo sobre el que se va a hacer una operación de lectura/escritura	*char*
objeto	Indica si la clasificación manual ha determindo que una imagen tiene objeto (1) o no lo tiene (0)	*char*
col	Siguiente columna que se va a modificar de la matriz *A*	*char*
frame	Información de la última imagen leída de un fichero	*cv::Mat*

Tabla 2: Tipo de dato variables de control del programa

Empezando por la parte offline, implementada dentro de una función para así poder llamarla de forma más sencilla cuando se quiera actualizar el fondo dinámico, las tres variables más importantes son: ‘A’, ‘U’ y ‘mean’ . ‘A’ contiene los valores de los píxeles de las imágenes que se usarán como fondo. En cada columna de esta matriz tengo una imagen. Debido a que esta se va a restar luego por su media y pueden quedarme valores positivos o negativos, no puede ser *unsigned*. Su rango de posibles valores será [-255,255], ya que es una imagen de un canal que usa 8 bits para la codificación del canal y pueden ser negativos o positivos ya que se le ha restado la media. Por lo que se ha optado por asignarle un tipo *short* (2 Bytes por dato). ‘U’ es la matriz que se usará como matriz de transformación en el algoritmo PCA. Los valores que contiene vienen de la multiplicación de una matriz de autovalores, que son números decimales, por la matriz ‘A’, que son enteros. Por lo que el resultado será un número decimal. Como se ha indicado anteriormente, cuando se necesite usar un número decimal se usará *float*, ya que es el tipo de dato en coma flotante que menos espacio ocupa y puede contener sin desbordar los datos que se le van a introducir. ‘mean’ guarda la media de los valores de los píxeles de las imágenes de fondo. Al ser una media, este valor también debería ser decimal, pero como se va a sumar y restar con arrays que tienen los píxeles de las imágenes y estos son enteros, no tiene sentido guardar un número decimal y luego al operar con él guardar un

entero. Por lo que este será de tipo *uchar* y así se guardan datos más pequeños y no se usan tantos recursos. Se usa *uchar* por no tener estos datos signo y estar comprendidos entre 0 y 255.

Dentro de la función que computa y muestra por pantalla la parte offline hay varias variables locales que hay que justificar el tipo de dato seleccionado para ellas. La primera que se usará es 'At'. Esta es la transpuesta de la variable 'A' de entrada a la función por lo que se pondrá del mismo tipo. Ya que los datos que va a guardar son los mismos que 'A' pero en otro orden. En cuanto a variable definidas como *float*, está 'Input', 'eigval', 'V', 'suma', 'total' y 'Utemp'. Los valores de la variable 'Input' pueden ser tanto positivos como negativos, ya que surge del producto de $A * A^t$ además sus valores pueden ser muy elevados aunque no sean decimales. Se ha estimado que para poder guardar estos valores en un entero sin que hubiese desbordamiento habría que usar un *long* (8 bytes). Pero como el *float* es más pequeño (4 bytes) y los valores que se obtienen de esta operación están dentro del rango de este tipo de dato, se opta por usar esta variable. Además, como en esta primera parte no hay ninguna función en HW, se puede usar números en coma flotante sin preocuparse porque el consumo de recursos HW se dispare mucho. En cuanto al resto de las nombradas usan o están compuestas por datos que se obtiene como salida del algoritmo SVD (función *dsvd* en el código). Por lo que tienen que ser números en coma flotante ya que la salida de este proceso son números en decimal y pueden ser tanto positivos como negativos. Las dos últimas variables que se pueden observar en esta función son 'i' y 'j', usadas como índices en los diversos bucles que hay en la función. Por lo que con usar enteros de tipo *int* es suficiente. Es verdad que al ser índices no van a ser nunca negativos y se podría usar *unsigned* pero se ha decidido dejar así ya que no se salen del rango. El uso de short también se ha descartado porque, sin signo, el valor de la variable desbordaría el rango por un valor pero que haría que la condición del *for* no se cumpliese nunca.

Variables parte offline		
Variable	**Descripción**	**Tipo dato**
A	Matriz con los valores de los píxeles de las imágenes de fondo	*short*
U	Matriz de transformación usada para pasar a espacio transformado. Ya solo con las columnas asociadas a los 'COMP' autovalores más representativos	*float*
mean	Valor medio de cada píxel de las imágenes de fondo	*uchar*
At	Matriz *A* transpuesta	*short*
Input	Matriz sobre la que se calcularán los autovectores y autovalores	*float*
eigval	Matriz de con autovalores de 'Input'	*float*
V	Matriz de autovalores de 'Input'	*float*
suma	Suma de los 'COMP' autovalores mayores	*float*
total	Suma de todos lo autovalores	*float*
Utemp	Matriz U completa	*float*
i	Índice de bucle	*int*
j	Índice de bucle	*int*

Tabla 3: Tipo de dato variables parte offline

En la parte online, hay un elevado número de variables. Las primeras que se pueden observar son las que se van a encargar de guardar las imágenes de entrada o salida del algoritmo('img','out' e 'img_dist') o las encargadas de guardar el mapa de distancias('MD'). Estas no van a tener que guardar valores negativos y todos van a estar entre 0 y 255, por lo que usar el tipo de dato *uchar*. Ya que es el tipo más pequeño que permite guardar los valores que toman estas variables sin desbordamiento. Las siguientes son las que guardan imágenes pero pueden tomar tanto valores negativos como positivos ('proyect_norm', 'recons_norm'). Por ello, se deberá asignar un tipo de dato que permita valores con signo y pueda almacenar el rango de valores [-255,255]. Entonces el tipo de dato necesario es *short*. La variable 'Ut' tiene que ser de tipo *float* ya que es la traspuesta de la variable 'U', por lo que deben tener el mismo tipo de dato. 'proyect' es también decimal porque es resultado de una multiplicación de un entero y un decimal y hay que tener la mayor precisión posible para que, a la hora de normalizar, se pueda realizar este proceso con la mayor exactitud posible. La variable 'porcentaje' guarda cuanto porcentaje de la imagen binarizada y procesada es blanco. Al ser un porcentaje que puede tomar valores decimales se opta porque el tipo de dato a usar sea el *float*. Con 'object' pasa lo mismo que la anterior porque es el valor que se va a usar de umbral de porcentaje para decidir si se ha detectado nuevo objeto o no.

A continuación hay dos variables definidas de una forma dinámica especial de SDSoC. Estas son 'imgmed' y 'recons'. Esta forma de asignar memoria de forma dinámica, aunque se le asigna un valor al inicio y ya no se cambia más en toda la ejecución, consiste en asignar un bloque de memoria física contigua del tamaño que se le indique. Para ello se usará *sds_alloc*. Esta difiere en la que se puede encontrar en C/C++ de que la memoria asignada no se hace partiendo de una dirección y asignando direcciones lógicas de memoria contiguas. Sino que la asignación se hace en memoria física contigua. Lo que hace que todo el array este guardado en un mismo dispositivo de memoria de todos lo que hay para guardar datos. De esta forma, a la hora de acceder a ellos, ya sea vía DMA o vía bus AXI, se pueda hacer estableciendo solo un bus para manejar los datos. Ya que si, aunque fuesen direcciones lógicas contiguas, estuviesen repartidas en dos dispositivos habría que usar dos buses distintos para poder acceder al array completo y no se podría usar entonces el pragma *SDS data zero_copy*.

Por último, están las variables de contabilización de errores y del número de imágenes procesadas en la ejecución del programa que serán enteros positivos siempre, por lo que se le pone un tipo de variable *unsigned*. Estas son: 'fpos', 'fneg' y 'img_proc'. Todas ellas inicializadas con el valor 0. Como los errores pueden ser muchos, se decide poner de tipo *short* para asegurarse que no desborda la variable y la contabilización se hace mal. La variable 'threshold' se ha puesto *unsigned* ya que no se puede tener una distancia negativa en este caso ya que se calcula como $|píxel_{input} - píxel_{output}|$ y esta operación no puede dar negativo. Como los valores pueden ser, como mucho, 255 se ha decidido poner un *char*. Por lo que el tipo de dato elegido es *uchar* para esta variable. Aquí también está definida la variable 'disp', que guardará la imagen binarizada para luego poderla guardar en un archivo y poder realizar los procesos de erosión y dilatación a este mapa de distancias binarizado. Por lo tanto, su tipo debe ser *cv::Mat.* Ya que este es un tipo de dato definido en las librerías OpenCV y es con el que trabajan todas las funciones de OpenCV cuando tiene que operar con una matriz. Este tipo de dato permite

guardar una imagen con el número de canales y de bits que se desee. Además tiene gran variedad de atributos como pueden ser *size* (tamaño de la matriz) o *reshape* (cambiar la forma de la matriz manteniendo el número de elementos). Lo que permite trabajar con las imágenes de una manera mucho más sencilla.

Variables parte online		
Variable	**Descripción**	**Tipo dato**
img	Matriz con los valores de los píxeles de la imagen de entrada	*uchar*
out	Matriz con los datos de la imagen de salida del algoritmo y sumado la media	*uchar*
MD	Mapa de distancias entre imagen entrada e imagen de salida	*uchar*
img_dist	Imagen resultante de binarización del MD	*uchar*
proyect_norm	Imagen proyectada normalizada	*short*
recons_norm	Imagen reconstruida normalizada	*short*
Ut	Matriz transpuesta de *U*	*float*
proyect	Imagen proyectada	*float*
porcentaje	Porcentaje de área blanca en imagen binarizada tras procesado de esta	*float*
object	Umbral para determinar si hay o no objeto en la imagen que se está procesando	*float*
imgmed	Imagen con media de los píxeles de fondo restada. Es φ del apartado 6.1.2 (parte online)	*short **
recons	Imagen reconstruida	*float **
fpos	Contador de falsos positivos	*unsigned short*
fneg	Contador de falsos negativos	*unsigned short*
img_proc	Contador de imágenes procesadas en la ejecución del programa	*unsigned short*
threshold	Umbral usado en la umbralización del MD	*uchar*
disp	Variable para guardar imagen en procesado para eliminar ruido y guardar en los ficheros	*cv::Mat*

Tabla 4: Tipos de datos variables parte online

En el caso de "multiplica_matrices.cpp", todas las variables locales a las funciones que hay en este archivo son enteros, de tipo *int*. Ya que se usan como índices de bucles para recorrer los matrices que se están multiplicando en cada una de las funciones.

Variables "multiplica_matrices.cpp"		
Variable	**Descripción**	**Tipo dato**
i	Índice bucle	*int*
j	Índice bucle	*int*
k	Índice bucle	*int*

Tabla 5: Tipos de datos variables "multiplica_matrices.cpp"

En "norm_err.cpp", las funciones de normalización (*normalize_R* y *normalize_P*) tienen una variable para recorrer el array de entrada, que será un *int* y una variable para guardar el máximo valor del array, que será de tipo *float* porque el array de entrada es de este tipo y esta variable alberga uno de esos valores. En el caso de *normalize_R*, como se va a implementar en HW, se ha añadido la variable 'val' para almacenar el valor de la posición del array que se está evaluando, si es mayor que el valor que tenía como máximo. Ya que si se hace una comparación directa con el valor de entrada sin pasarlo por una variable intermedia la función se implementa mal y no funciona. Ya que este array no se copia en HW y no se puede acceder a cualquier posición de memoria siempre que se desee, ya que el comparar con un valor guardado en memoria compartida, ralentiza mucho el programa porque hay que hacer muchos accesos a memoria. De esta forma, se lee todo seguido el array y no hay que hacer saltos en memoria en cada iteración que provocan pérdidas de tiempo y el acceso a él es más rápido y comparar con el valor de un registro local es mucho más rápido y fácil de implementar.

La función *recons_error* tiene dos variables, que son el índice del bucle y la variable que almacena el valor de la diferencia entre el píxel reconstruido y el original. Las dos solo van a almacenar valores positivos, así que se elegirán como *unsigned*. El índice del bucle es de tipo *int*, ya que con el short desborda y el bucle no acabaría. La variable que guarda la diferencia entre uno y otro valor, 'diff', será de tipo *char*, ya que guarda la diferencia de dos valores *char*. Por lo que no necesita más rango que el que dan esos datos. Además luego ese valor se copiará a una posición de un array de este mismo tipo.

Las siguientes funciones para explicar son *threshold_img* y *ordena_matrix*. La primera solo tiene dos variables que se usarán como índices y son de tipo *short*, ya que no pasarán del valor 256. La segunda tiene 4 variables locales. 'temp' funcionará de variable intermedia para mover los datos en las matrices de los autovectores y los autovalores. Por lo que tendrá que ser del mismo tipo de dato (*float*) para que no haya problemas al asignar un elemento del array a esta variable o viceversa. Luego están los índices de los bucles que son *unsigned char*. Al igual que la variable usada como flag para saber cuándo el proceso de ordenar los autovalores ha concluido ('mod').

Variables "norm_err.cpp"		
Variable	**Descripción**	**Tipo dato**
max	Máximo valor , en valor absoluto, del array de entrada	*float*
val	Valor absoluto del píxel que se está procesando	*float*
i (*normalize*)	Índice del bucle	*int*
diff	Diferencia de valor en el píxel que se está evaluando entre la imagen de entrada y la de salida del algoritmo	*unsigned char*
i (*recons_error*)	Índice del bucle	*unsigned int*
i (*threshold_img*)	Índice del bucle	*short*
j (*threshold_img*)	Índice del bucle	*short*
temp	Variable temporal para poder hacer intercambio de valores entre dos posiciones sin perder datos	*float*

mod	Comprobar si ha habido alguna modificación en la última iteración del *while* dentro de *ordena_matrix*	*unsigned char*
i (*ordena_matrix*)	Índice de bucle	*unsigned char*
j (*ordena_matrix*)	Índice de bucle	*unsigned char*

Tabla 6: Tipos de dato variables de "norm_err.cpp"

El último archivo realizado para este trabajo es "transpose_mean.cpp". En él, todas la variables de las funciones son *int*, ya que son límites de bucles que se usarán para ir recorriendo cada uno de los arrays e ir haciendo las operaciones para las que cada función está destinada. A excepción de 'suma' en *mean_row*, que no es un índice. Sino que se usa para ir acumulando la suma de cada una de las filas de la matriz A para luego poder sacar la media de cada una de ellas. Esta es de tipo *short*, ya que este es el primer tipo de dato que no desborda el rango de este tipo de variables ni en el peor de los casos.

Variables "transpose_mean.cpp"		
Variable	**Descripción**	**Tipo dato**
i	Índice bucle	*int*
j	Índice bucle	*int*
suma	Acumula valor de la suma de todos los píxeles i-ésimos de las imágenes que se usan de fondo	*short*

Tabla 7:Tipos de datos de las variables de "transpose_mean.cpp"

Es verdad que en alguna de las funciones que se ha comentado anteriormente el tipo de dato elegido para la variable puede que no sea el óptimo en cuanto a uso de recursos, aunque en las funciones que se van a implementar en HW si se ha conseguido asignar el tipo de dato más ajustado a los valores que almacena la variable. En SW no es tan relevante porque no hay que ahorrar al máximo el uso de recursos de la FPGA para poder implementar lo máximo que se pueda en HW.

6.5 Optimización HW

La forma con la que hay que proceder para conseguir hacer que el consumo de recursos usados para implementar una función sea mínimo y que el tiempo que tarda en desempeñar su objetivo sea mínimo también, es a través de la incorporación se sentencias pragma en el código original. Estas sentencias son indicaciones que se le dan al compilador de SDSoC para que implemente en HW las secciones de código o las variables a las que hace referencia como el desarrollador considere más oportuno. En caso de no poder conseguir el mínimo en los dos factores que determinan cómo de bueno o de malo es la implementación a la vez, hay que llegar a una solución de compromiso. En ella, hay que primar la latencia menor sin que el consumo de recursos se dispare de forma desorbitada. Hay dos tipos de sentencias que se usarán: las que se encuentran en SDSoC (aquellas que comienzan con SDS) y las que proporciona HLS (estas comienzan con HLS). Para poder incluir las segundas y ver su impacto en el rendimiento de cada una de las funciones que se van a pasar a HW, habrá que lanzar el programa HLS desde la opción  en el archivo "project.sdx".

Para que el programa se pueda compilar de forma correcta y sin errores lo primero que hay que hacer, usando las directivas SDS, es indicarle al programa cómo va a ser el acceso a cada una de las variables de entrada/salida de la función. Si no se indica nada, el SDSoC va a intentar copiar todas las variables que se necesitan para las funciones aceleradas a BRAM y, como se está trabajando con variables muy grandes dado que son imágenes, va a dar error de espacio insuficiente para ubicarlas. Por ello, se han empleado dos estrategias para solventar este problema basándose en el tipo de acceso que se necesita de cada una de estas variables:

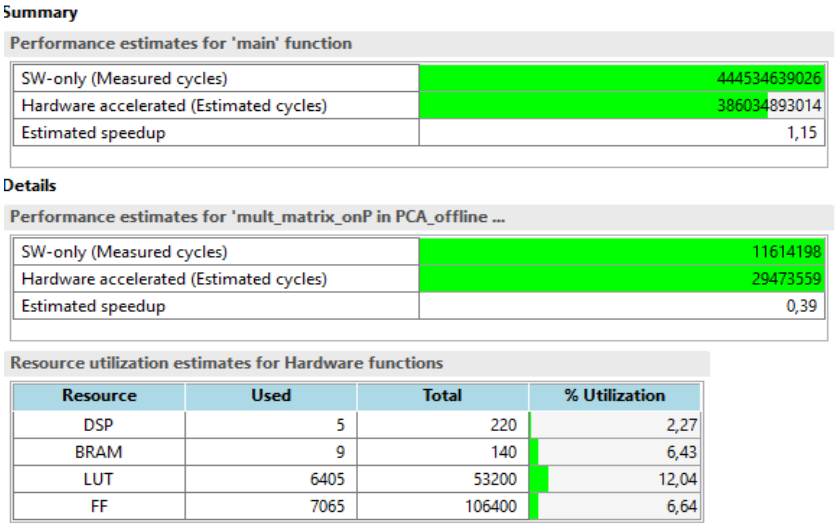
- **FIFOs:** Las variables cuyo acceso a ellas se hace de forma secuencial, como puede ser una array que se va a ir accediendo cada vez a la posición siguiente a la usada en la anterior iteración. Por ejemplo, el array donde se guardan los valores entre 0 y 255 de la imagen reconstruida en "normalize_R". Se incluirá una sentencia pragma para que su acceso se haga a través de una FIFO con DDR y no se copien en BRAM. Esta se introducirá inmediatamente anterior a la declaración de la función. La directiva usada será "#pragma SDS data access_pattern (<variable>:SEQUENTIAL)". Así podrá saber que su acceso es secuencial y es fácilmente trasladable a, en este caso, enviar los datos a/desde DDR a través de una FIFO. Para poder aplicar esta solución, además de acceder de forma secuencial, solo se tiene que acceder una vez a cada posición del array en cada llamada a la función. Ya que si se recorre varias veces, el salto del final al principio no es un desplazamiento a una posición contigua sino que es un salto en memoria no permitido por esta filosofía de acceso a memoria.
- **Acceso aleatorio:** En el caso de las multiplicaciones de matrices, la matriz que se emplea como segundo operando, se recorre de forma secuencial pero varias veces, por lo que hay saltos en memoria más grandes que a posiciones contiguas. Así que hay que buscar otra forma de indicarle al programa que realice una serie de acciones para evitar que lo aloje todo en BRAM pero se pueda leer correctamente todos los datos igualmente. En concreto, hay que realizar dos cosas. La primera de ellas es poner la sentencia "#pragma SDS data zero_copy(<nombre_variable>[0:<tamaño_variable>])". Esto indica que los datos para la función HW se leen/escriben directamente desde la memoria compartida con la parte programable, RAM. Sin hacer una copia de ellos para la función hardware. La forma de acceder a ellos será a través de una interfaz de bus AXI, el compilador elegirá la versión que mejor se adapte a la forma de acceder a los datos. Además de incluir esta sentencia, las variables sobre las que se vaya a usar esta sentencia se tienen que alojar en memoria a través de la función *sds_alloc*, contenida en la librería "sds_lib.h". Esta función permite alojar en posiciones contiguas de memoria física el tamaño del array que se le indique. De esta forma a la hora de acceder al array con tener un solo bus AXI vale. Ya que toda la variable estará dentro del mismo dispositivo de memoria. Es verdad que se comentó anteriormente que se debería evitar a toda costa el uso de memoria dinámica. Pero como esto es una asignación que

se hace al inicio del programa y no se modificará su tamaño en ningún punto, el cual se sabe en tiempo de compilación cuál es. Esto se puede considerar como si se declarase un array de un cierto tamaño en dos pasos. Si no se usase este comando, se puede dar el problema de que no todo el array se alojase en un solo dispositivo de memoria, ya que hay varias pastillas en la Zedboard, aunque en el mapa de memorias lógicas del sistema ocupase posiciones contiguas. Esto haría que se tuviese que establecer dos bus AXI para poder gestionar todo el array ya que, físicamente, no está todo junto y esto aumenta la complejidad de la implementación considerablemente. Ya que habría que ver qué bus AXI usar para recibir los datos que se necesitan en cada momento según en la posición del array que se esté usando Este aspecto SDSoC no lo puede solucionar y da un error de compilación. Hay que tener en cuenta que estos son punteros a los que se les asigna memoria durante la ejecución del programa. Por lo que al finalizar hay que liberar esa memoria con *sds_free*. Este tipo de variables se pueden poner en el pragma *SDS data access pattern* como acceso ramdon.

Estas consideraciones se han tenido en cuenta para las variables: ‘Ut’, ‘imgmed’, ‘U’, ‘recons’ y ‘recons_norm’. Debido a su gran tamaño, que impedía que el programa las copiará en BRAM para su uso, opción por defecto que intenta implementar el SDSoC. Ya que no tenía suficiente memoria como para poder copiarlas todas las que necesitaba para cada función.

Tras haber configurado correctamente la entrada y la salida de las funciones HW. Se puede avanzar al siguiente paso: la optimización en HLS. Para ello, habrá que ir seleccionando cada una de las funciones que se han incluido para su aceleración e ir entrando en HLS, como se indica en el inicio de este apartado, con cada una de ellas para poder ir estudiando caso a caso qué directivas permiten tener una implementación lo más optimizada posible. Como se comentó con anterioridad se dará prioridad a la reducción de la latencia, en caso de que la reducción sea mínima, se mirará cómo varía el uso de recursos para ver si interesa o no. Ya que la limitación de recursos no es un problema porque no se va a llenar la placa ni a llegar a niveles de ocupación elevados.

La primera función que se va a someter a esta optimización es *mult_matrix_onP*. Antes de saltar a HLS, se va a seleccionar solo esta como función a acelerar y se va a ejecutar la opción de *Performance Estimation*. Lo que se extrae de ejecutar esta opción es los ciclos que se estima que consume el HW acelerado y el uso de recursos que supondrá el paso de la función a HW. Los ciclos son del reloj que use la CPU, en este caso 666,666687 MHz. Además se muestra tanto la aceleración del algoritmo completo al mover la función a HW, como la aceleración de la función aislada al trasladarse a HW. Los resultados obtenidos son los mostrados en la siguiente figura.



Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	444534639026
Hardware accelerated (Estimated cycles)	386034893014
Estimated speedup	1,15

Details

Performance estimates for 'mult_matrix_onP in PCA_offline ...

SW-only (Measured cycles)	11614198
Hardware accelerated (Estimated cycles)	29473559
Estimated speedup	0,39

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	5	220	2,27
BRAM	9	140	6,43
LUT	6405	53200	12,04
FF	7065	106400	6,64

Figura 18: Performance Estimation mult_matrix_onP

Como se puede empezar a observar en esta primera implementación el mayor consumo de recursos se va a producir en las LUT y el menor en DSP, a pesar de usar multiplicaciones. También queda patente que se necesita una optimización de HW ya que la aceleración en este caso es menor de 1, lo que indica que la función, en esta situación, tarda más en ejecutarse en HW que si se dejase en SW, cosa que no interesa. A pesar de este aumento en la latencia, la aceleración es mayor que 1. Esto indica que se consigue acelerar el algoritmo global. La explicación de este hecho es que el SDSoC, al introducir una función en HW, paraleliza su procesado con el del SW, lo que hace que todo vaya más rápido ejecutándose, por estar haciendo varias cosas en paralelo, y el tiempo de proceso total sea menor.

Con esta primera estimación se puede dar el salto a HLS para encontrar la solución óptima de implementación de esta función. Para ello, hay que ir probando cuales de todas las posibles sentencias pragma de las que dispone HLS puede hacer que la latencia baje considerablemente sin que el consumo de recurso se dispare en gran medida. En este caso, la directiva que se ha usado es *PIPELINE* con la opción de *rewind*. Esta opción permite que no haya parada entre una iteración y la siguiente. De esta forma la latencia baja considerablemente ya que se consigue tener varias operaciones a la vez ejecutándose lo que provoca que el tiempo consumido para realizar la funcionalidad total disminuya bastante. Este paralelismo provoca un aumento de recursos que se decide asumir, ya que la mejora de tiempo es muy grande, reducción a en torno a un tercio de la latencia inicial, a costa de un aumento de recursos de bastante reducido, 119 LUT y 27 FF más. Con la opción *rewind* de la sentencia se ahorran 5 ciclos de reloj y el número de recursos consumidos totales no cambia. Todo esto contemplado en la Figura 20. Con este "*fine tunning"* se ha conseguido mejorar bastante el tiempo de ejecución de la función. Eso contribuirá a que el *speed-up* final sea más grande, lo que provoca menor latencia de ejecución del algoritmo.

Latency (clock cycles)

		incio	pipeline	pipeline_rewind
Latency	min	4194337	1310805	1310800
	max	4194337	1310805	1310800
Interval	min	4194337	1310805	1310800
	max	4194337	1310805	1310800

Utilization Estimates

	incio	pipeline	pipeline_rewind
BRAM_18K	2	2	2
DSP48E	5	5	5
FF	1467	1590	1586
LUT	2198	2221	2225

Figura 20: Latencia y uso de recursos de cada una de las opciones tomadas en mult_matrix_onP

Tras haber conseguido optimizar la latencia lo máximo que se ha podido, se incluyen las directivas en el código y se guarda el fichero. A continuación, se cierra HLS y se vuelve a lanzar el *Performance Estimation* para ver la mejora de velocidad. Lo que ahora devuelve indica que el consumo de recursos ha aumentado debido a lo comentado anteriormente y que la latencia ha bajado, menos ciclos de reloj para ejecutarse. Esto queda recogido en la Figura 19. El aumento de recursos entre HLS y SDSoC se debe a que HLS solo tiene en cuenta los usados por la función que se abre para optimizar. En cambio, SDSoC contempla los recursos que necesita para implementa los buses necesarios para conectar la función HW con memoria como se enunció anteriormente e implementar todo lo solicitado en las sentencias pragma de SDS, que HLS no las procesa. Se puede observar que ahora el *speed-up* de la función es mayor que 1, lo que indica que si es interesante pasarla a HW. Aunque la aceleración total no se ha modificado, se dejarán las mejoras incluidas, ya que al introducir más funciones la aceleración de cada función irá decayendo. Al no ser el consumo de recursos un factor limitante en este caso, como se verá más adelante, se opta por dejar las funciones lo más optimizadas posible aunque esto suponga un aumento de recursos que no repercutirán en la mejora de la aceleración final.

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	444534639026
Hardware accelerated (Estimated cycles)	385652795176
Estimated speedup	1,15

Details

Performance estimates for 'mult_matrix_onP in PCA_offline ...

SW-only (Measured cycles)	11614198
Hardware accelerated (Estimated cycles)	10249978
Estimated speedup	1,13

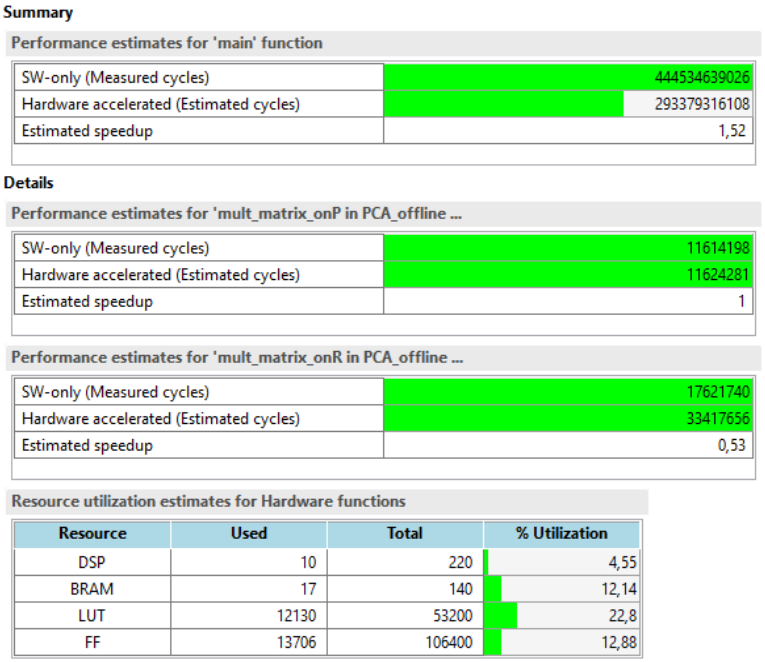
Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	5	220	2,27
BRAM	9	140	6,43
LUT	6432	53200	12,09
FF	7184	106400	6,75

Figura 19: Resultado Performance Estimation después de pasar por la optimización en HLS de mult_matrix_onP

Tras acabar con la optimización de esta función, se pasa a optimizar la función *mult_matrix_onR*. Esta también consume mucho tiempo de CPU si se procesa en SW así que se espera poder conseguir una reducción tan buena como la obtenida para la anterior. Ahora se lanza el *Estimate Performance* incluyendo esta nueva función para ver como los recursos consumidos en la FPGA son mayores y así tener un control de cuando hay que parar de introducir más funciones en HW por tener esta parte muy llena. En este caso no se llega a una situación en la que hay que elegir la función que se queda fuera de HW aunque consuma mucho en SW por quedarse sin recursos.

Un aspecto interesante que se puede extraer de la Figura 21 es que el tiempo de ejecución estimado de la función ya optimizada aumenta al introducir una nueva en HW. Esto está provocado porque, al tener más funciones en la parte HW, no se puede implementar todos los recursos de un bloque de forma que se consiga tiempos de propagación mínimos entre las distintas partes de este, lo que provoca que las funciones HW sean más lentas. Esto indica que tampoco es bueno llenar la FPGA hasta valores cercanos al 100%, ya que puede que la latencia de las funciones al tener muchos bloques HW metidos en la FPGA aumente mucho y las optimizaciones realizadas no tengan tanto efecto como se desearía.



Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	444534639026
Hardware accelerated (Estimated cycles)	293379316108
Estimated speedup	1,52

Details

Performance estimates for 'mult_matrix_onP in PCA_offline ...

SW-only (Measured cycles)	11614198
Hardware accelerated (Estimated cycles)	11624281
Estimated speedup	1

Performance estimates for 'mult_matrix_onR in PCA_offline ...

SW-only (Measured cycles)	17621740
Hardware accelerated (Estimated cycles)	33417656
Estimated speedup	0,53

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	10	220	4,55
BRAM	17	140	12,14
LUT	12130	53200	22,8
FF	13706	106400	12,88

Figura 21: Performance Estimation al añadir mult_matrix_onR

Ahora, al igual que antes, se llama de nuevo HLS pero ahora seleccionando la función “mult_matrix_on_R”. En este caso, y debido a que el cuerpo de la función es el mismo que el de la función anterior, se ha optado por aplicar la misma solución de optimización que en la anterior función: sentencias *PIPELINE*. Sin embargo, en este caso, no solo en el bucle más interno sino que también se ha introducido en el más externo. Esto es así ya que el bucle más interno en este caso es pequeño, solo ‘COMP’ (vale 4 en el caso del trabajo) iteraciones, y hay recursos disponibles para poder paralelizar los dos bucles. De esta forma se ha conseguido reducir en gran medida el tiempo de ejecución de

esta función llegando a quedarse alrededor de un 5% de la latencia inicial. La optimización aquí ha conseguido grandes resultados, ya que se han incluido dos sentencias *PIPELINE* en vez de una a costa de aumentar el consumo de recursos considerablemente, sobre todo en el número de FF usados (se pasa de 924 a 1522). En la Figura 22 también se puede ver que la opción *rewind* en este caso no es interesante ya que baja la latencia 2 ciclos pero el consumo de recursos aumenta considerablemente. Por lo que no sería optimo tomar esta opción, ya que no compensa el aumento de recursos la reducción de latencia.

Latency (clock cycles)

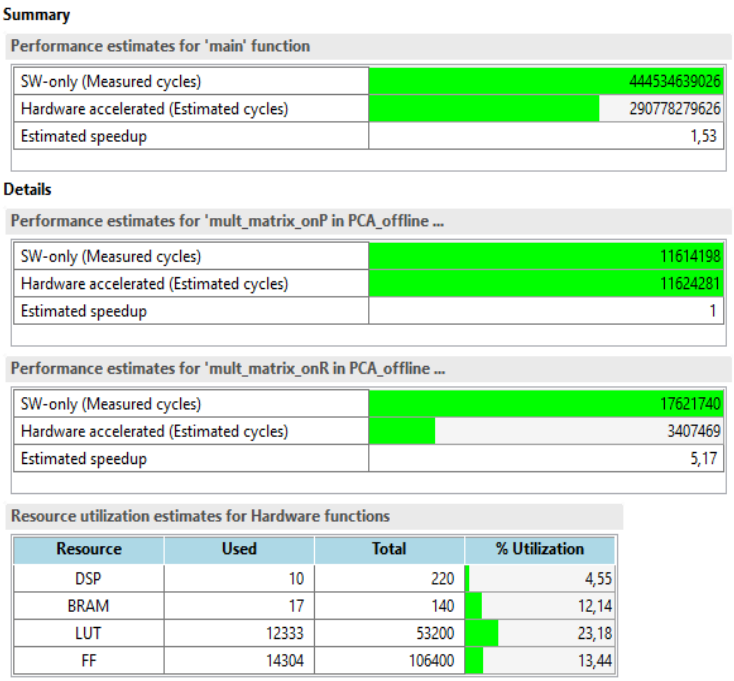
		inicio	pipeline_loop3	pipeline_loop31	pipeline_loop31_rewind
Latency	min	4849665	2293761	262174	262172
	max	4849665	2293761	262174	262173
Interval	min	4849665	2293761	262174	262144
	max	4849665	2293761	262174	262144

Utilization Estimates

	inicio	pipeline_loop3	pipeline_loop31	pipeline_loop31_rewind
BRAM_18K	0	0	0	0
DSP48E	5	5	5	5
FF	924	981	1522	2128
LUT	1491	1501	1694	1994

Figura 22: Latencia y consumo de recursos de cada una de las opciones tomadas para la optimización de la función mult_matrix_onR

En este caso la reducción del tiempo de proceso es más grande. Antes era de un tercio aproximadamente y ahora es de en torno a un 5,4% de lo inicial. Después se debe cerrar HLS asegurándose que las sentencias están introducidas en el código y corresponden con la solución adoptada, para poder volver a llamar al *Estimate Performance*. Con ello, se obtendrá el nuevo tiempo de ejecución de la función optimizada. Se puede observar el aumento de recursos en la FPGA y la reducción de los ciclos de proceso en la Figura 23. Esta queda presente en que se ha multiplicado casi en 10 veces la aceleración de la función. En cambio, la aceleración global no ha crecido tanto, solo un 0,4 más. Esto muestra que, si se intentan acelerar funciones con menos tiempo de CPU, la aceleración global no es tan grande como la que se puede conseguir en la aceleración de la función, justificado empíricamente así lo expuesto anteriormente. Esto será más notable según la función consume menos tiempo de CPU en el conjunto total de la ejecución del programa. Lo obtenido hasta el momento es lo mostrado en la siguiente figura.



Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	444534639026
Hardware accelerated (Estimated cycles)	290778279626
Estimated speedup	1,53

Details

Performance estimates for 'mult_matrix_onP in PCA_offline ...

SW-only (Measured cycles)	11614198
Hardware accelerated (Estimated cycles)	11624281
Estimated speedup	1

Performance estimates for 'mult_matrix_onR in PCA_offline ...

SW-only (Measured cycles)	17621740
Hardware accelerated (Estimated cycles)	3407469
Estimated speedup	5,17

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	10	220	4,55
BRAM	17	140	12,14
LUT	12333	53200	23,18
FF	14304	106400	13,44

Figura 23: Performance Estimation tras optimización de mult_matrix_onR

En este punto se puede decir que ya se lleva la mayor parte de la aceleración realizada. Porque, aunque todavía quede una de las funciones por optimizar, sobre las que ya se ha realizado este proceso de optimización son las que, según mostró el *TCF Profiler*, las que más tiempo de CPU consumían (un 43,6% entre las dos). Por tanto estas aceleraciones son las que más van a contribuir al *speed-up* de la aplicación global.

Tras esta pequeña reflexión se continúa con el proceso de optimización. Ahora, para concluir, se va a realizar el proceso de optimización de la última función que se ha decidido pasar a HW debido a su alto consumo de tiempo de CPU. Esta es la función *normalize_R*. Debido a que en ella se normaliza con respecto al máximo valor del array y luego se multiplica por 255, esto conlleva un gran cómputo en software, ya que hace uso de muchas multiplicaciones y divisiones. Al pasarlo a HW, se traduce en gran uso de recursos. El mayor aumento se produce en el uso de FF (más de 7500 solo para implementar esta función sin optimizar aún) y en el de LUT (unas 7200). Esto queda patente en la Figura 24, realizada tras introducir la función para acelerarla en HW y ejecutar *Estimate Performance*.

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	444534639026
Hardware accelerated (Estimated cycles)	259492342048
Estimated speedup	1,71

Details

Performance estimates for 'mult_matrix_onP in PCA_offline ...

SW-only (Measured cycles)	11614198
Hardware accelerated (Estimated cycles)	12631459
Estimated speedup	0,92

Performance estimates for 'mult_matrix_onR in PCA_offline ...

SW-only (Measured cycles)	17621740
Hardware accelerated (Estimated cycles)	3407469
Estimated speedup	5,17

Performance estimates for 'normalize_R in PCA_offline.cpp ...

SW-only (Measured cycles)	6037666
Hardware accelerated (Estimated cycles)	18370005
Estimated speedup	0,33

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	13	220	5,91
BRAM	26	140	18,57
LUT	19565	53200	36,78
FF	21869	106400	20,55

Figura 24 : Performance Estimation al añadir normalize_R

Tras ver el consumo de recursos estimado que habrá tras introducir las 3 funciones en HW, se pasa a optimizar la función *normalize_R* para conseguir la máxima velocidad. En este caso, las sentencias pragma usadas para conseguir esta optimización a la hora de implementar la función son el uso de dos *PIPELINE*, uno en cada uno de los dos bucles que la componen. De esta forma, se consigue reducir en gran medida el tiempo de proceso, pasando de casi 2,3 millones de ciclos por cada vez que se llama a la función a solo unos 130.000 ciclos. Esto supone una reducción de la latencia de unas 17,5 veces, como se puede observar en la Figura 25. El consumo de recursos es elevado desde el principio ya que es una función en la que se usan instrucciones que necesitan muchos recursos aunque con las distintas opciones que van haciendo mejor el rendimiento no aumenta mucho el consumo de recursos. La opción *rewind* no se usa ya que no se consigue un ahorro de latencia muy grande, aunque si se ahorran recursos. Pero como ya no se van a introducir más funciones y hay mucho espacio libre, como se verá a continuación no se ha considerado necesario reducir más los recursos.

Latency (clock cycles)

		Inicio	pipeline_max	pipeline_norm	pipeline_all
Latency	min	2293765	1638424	786461	131120
	max	2293765	1638424	786461	131120
Interval	min	2293765	1638424	786461	131120
	max	2293765	1638424	786461	131120

Utilization Estimates

	Inicio	pipeline_max	pipeline_norm	pipeline_all
BRAM_18K	2	2	2	2
DSP48E	3	3	3	3
FF	1967	1982	2019	2034
LUT	3025	3051	2995	3021

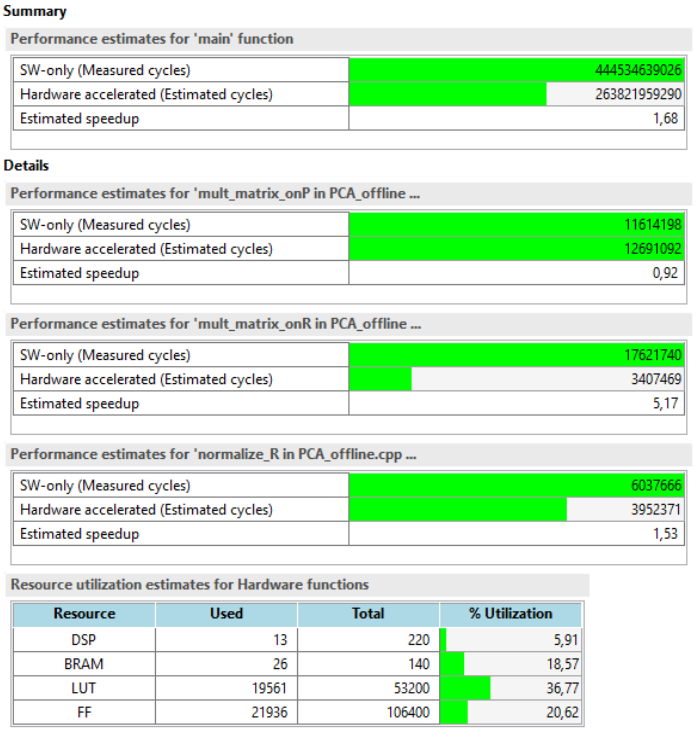
Figura 25: Consumo de recursos y latencia en HLS de la función normalize_R

Elegidas las directivas a usar e introducidas en su correspondiente sitio en el código, se procede a cerrar el HLS y realizar un último *Estimate Performance* con ya todas las funciones optimizadas y puestas para acelerar en HW. Como se puede observar en la Figura 25, el consumo de LUT ha bajado y el de FF ha subido en una pequeña proporción. Esto no es muy llamativo en comparación con la gran reducción conseguida en el tiempo de proceso de la función *normalize_R*, que se queda en torno al 5,72% de la latencia que había antes de este proceso de optimización. Esta gran reducción afectará, de forma positiva, al rendimiento final del programa.

Por lo tanto, con esto se puede decir que el proceso de optimización y selección de las funciones que se van a introducir en HW ha concluido. El consumo de recursos en la FPGA final será de: 13 DSP, 26 bloques BRAM, 19561 LUT y 21936 FF. Siendo el porcentaje de uso de LUT el mayor de todos, seguido de FF y BRAM. El porcentaje menor son de DSP.

El *speed-up* conseguido total es de 1,68. Esto quiere decir que el algoritmo usando una combinación de SW y HW tarda en completarse esas veces menos que si se hiciese todo en SW. Por lo que se puede decir que el ahorro de tiempo conseguido es bastante bueno, ya que, si se observan los números de ciclos estimados de uno y otra forma de ejecutarse, se consigue reducir la ejecución en unos 180,7 mil millones de ciclos. Estos ciclos se miden a la frecuencia de reloj de la CPU, unos 667MHz. Por lo que se va a obtener un ahorro de alrededor de 4 minutos y medio. En cuanto al análisis función por función, en la que más *speed-up* se consigue es *mult_matrix_onR*. En esta se consigue meter dos *PIPELINE* que hacen que la aceleración sea máxima ya que se consigue paralelizar al máximo las distintas operaciones que tienen lugar dentro de ella. Es verdad que *normalize_R* tiene también dos sentencias *PIPELINE* pero como son de dos bucles independientes no se consigue que el efecto sea tan bueno porque hay que ejecutar uno y luego el otro, y no se hace los dos a la vez, por depender los datos de entrada en el segundo de lo que se procesa en el primero. En cambio, en *mult_matrix_onP* el speed up es menor que 1, pero, como es una función que se ejecuta muchas veces y al estar en HW se puede paralelizar con el procesado de otras funciones en SW, el efecto sobre el total es de reducción de la latencia. Si se quitase esta última, la aceleración total del proceso baja

considerablemente. Así que interesa dejarla en HW aunque se estime que no está acelerada si se mira de forma independiente. También se ha observado que la aceleración global disminuye en una pequeña medida respecto a sin optimizar *normalize_R*. Pero se ha decidido dejar así para ver que un proceso de optimización de una función puede reducir el tiempo consumido por esta pero aumentar el global y así poder tener tres procesos de optimización de distintas funciones.



Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	444534639026
Hardware accelerated (Estimated cycles)	263821959290
Estimated speedup	1,68

Details

Performance estimates for 'mult_matrix_onP in PCA_offline ...

SW-only (Measured cycles)	11614198
Hardware accelerated (Estimated cycles)	12691092
Estimated speedup	0,92

Performance estimates for 'mult_matrix_onR in PCA_offline ...

SW-only (Measured cycles)	17621740
Hardware accelerated (Estimated cycles)	3407469
Estimated speedup	5,17

Performance estimates for 'normalize_R in PCA_offline.cpp ...

SW-only (Measured cycles)	6037666
Hardware accelerated (Estimated cycles)	3952371
Estimated speedup	1,53

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	13	220	5,91
BRAM	26	140	18,57
LUT	19561	53200	36,77
FF	21936	106400	20,62

Figura 26: Consumo de recursos y latencia tras introducir todas las funciones a acelerar.

Ahora se incluirá una tabla que servirá para ilustrar cómo este proceso de optimización ha conseguido reducir los tiempos de proceso para cada una de las funciones introducidas a costa de aumentar los recursos usados. Tanto el porcentaje de tiempo que tarda tras la optimización con respecto al que tardaba antes de ella como el aumento de recursos, se extraerán de los informes devueltos por SDSoC ya que estos representan más fielmente lo que se implementará en placa finalmente.

En ella, se puede observar que las mejoras introducidas no provocan, en ningún caso, el aumento de recursos de DSP y BRAM ya que la sentencia *PIPELINE* introducida hace que introduzcan registros intermedios para poder tener circuitos más sencillos que procesan en paralelo datos en vez tener uno grande que haya que esperar hasta que procese todo para poder introducirle otro dato. El tiempo medio de proceso de las funciones tras la optimización es del 22% del tiempo original.

Función HW	Optimización	Aumento recursos	% de tiempo original
mult_matrix_onP	Sentencia *PIPELINE* en bucle más interno con *rewind*	DSP: 0 BRAM: 0 LUT: 27 FF: 119	34,78%
mult_matrix_onR	Sentencia *PIPELINE* en bucle más interno y en bucle más externo sin *rewind*	DSP: 0 BRAM: 0 LUT: 203 FF: 598	10,2%
normalize_R	Sentencia *PIPELINE* en cada uno de los bucles sin *rewind*	DSP: 0 BRAM: 0 LUT: -4 FF: 67	21,52%

Tabla 8:Comparativa de tiempo y recursos consumidos debido a someter a las funciones HW a proceso de optimización.

6.6 Resultados

Una vez realizado el proceso de optimización y haber obtenido cada uno de los *speed-up* estimados de cada una de las funciones que hay en HW y cómo afectan cada una de ellas al *speed-up* del proceso en general, se pasa a presentar los resultados de tiempo de proceso tanto para todo SW como para el particionado HW/SW. Debido a que se querían hacer medidas de varias ejecuciones del programa, se decidió trocear el set de imágenes en tres partes, cada una con 1794 imágenes, y así hacer que las ejecuciones fuesen más cortas y tener más datos para poder hacer una media de más datos. Hay que comentar que las imágenes que formarán el fondo inicial son las mismas en los tres casos, las 8 primeras del conjunto. De cada una de ellas, se realizó 5 ejecuciones para solo SW y otras 5 para el programa particionado. La forma de medir el tiempo de ejecución fue haciendo uso del comando *time* que se puede encontrar dentro del SO empotrado en el SoC. Este devuelve tres tiempos, de los cuales son interesantes dos: real y user. El de real (tiempo que transcurre desde que se lanza el programa hasta que concluye) es el que se comparará con las estimaciones del SDSoC sobre aceleraciones y tiempos de ejecución ya que este es el tiempo de ejecución con toda la sobrecarga que introduce el sistema operativo. El de user (tiempo que se está ejecutando realmente el programa) se observará ya que es interesante tenerlo en cuenta porque puede dar una idea de lo que tardaría en ejecutarse el programa si fuese *standalone*. Además en él se podrá ver, de forma clara, cómo al introducir mejoras el tiempo de proceso aumenta.

Tras tomar todas las medidas, se hace la media de los tiempos de ejecución de todas las iteraciones para cada parte, diferenciando entre user y real. Una vez se tiene la media de cada parte se hace la media de las tres partes y este será el valor que se toma como de ejecución medido. Si este se multiplicase por 3, número de partes del programa,

se obtendría el tiempo total de proceso. En las tablas se muestra el valor del tiempo en segundos para la ejecución de un tercio del programa.

Los parámetros que se han considerado interesantes medir son:

- **El tiempo de la parte offline:** este tiempo es interesante medirlo para ver cuanto tarda el sistema en empezar a procesar imágenes de entrada. Además, también interesará saberlo para ver cuánto tiempo necesita cada vez que se quiera actualizar el fondo para poder seguir procesando imágenes. Ya que no se hace este cálculo de forma paralela, a priori.
- **El tiempo en la parte online:** este se obtendrá como la resta del total menos el tiempo que está en offline y me servirá para ver el tiempo que tarda la aplicación en procesar todas las imágenes que se le dan. La fórmula para calcularlo sería la siguiente:

$$T_{online} = T_{total} - T_{offline} * n_{ejecuciones\ \ parte\ offline}$$

- **El FPS:** es el número de frames por segundo que puede procesar el sistema. Es interesante saberlo porque nos da una idea de lo rápido que puede la aplicación procesar información. Este se calculará tanto para el procesado solo SW como el particionado de HW/SW. Este se calculará como: $FPS = \frac{N_{frames\ procesados}}{T_{total}}$.
- **La aceleración:** mide cuánto más rápido es el procesado HW/SW que el solo SW. También llamado *speed-up*. Este es el más interesante de todos ya que es el que se está intentando maximizar en todo momento con la optimización, a través de pragmas, del HW. Esta se comparará con lo estimado por SDSoC para ver si la estimación dada en un primer momento es buena o mala. Cuanto mayor sea este valor mejor se habrá hecho el proceso de optimización y elección de las funciones que se mueven a HW y las que se dejan en SW.

Tras unas pequeñas aclaraciones de lo que se espera obtener en esta parte, se pasa a mostrar los resultados obtenidos tras hacer las medias de los tiempos de ejecución.

Lo primero de todo es ver el tiempo que tarda en ejecutarse la parte offline para poder restarla al tiempo que está ejecutándose la parte online. Así se podrán hacer los cálculos de FPS teniendo en cuenta solo el tiempo que ha estado procesando realmente las imágenes. Tras realizar varias medidas, el tiempo medio que tarda en procesarse la parte online es:

Parte offline	real	user	sys
Media (s)	1,135	1,02	0,115

Tabla 9: Tiempos de cómputo parte offline

En estas primeras medidas se ejecutó el algoritmo sin la posibilidad de que el fondo se fuese actualizando con imágenes de entrada cada cierto tiempo. En este caso los resultados obtenidos son:

Parte online SW 1794 imágenes	real	user	sys
Medida 1 (s)	255,026	156,475	46,959
Medida 2 (s)	294,181	156,755	84,781
Medida 3 (s)	197,701	156,497	19,949
Media (s)	248,969	156,576	50,563
FPS	7,206	11,458	

Tabla 10: Tiempos medios de proceso solo SW para cada uno de los sets de imágenes

Parte online SW/HW 1794 imágenes	real	user	Sys
Medida 1 (s)	162,102	54,873	46,246
Medida 2 (s)	194,210	54,485	81,148
Medida 3 (s)	76,661	54,010	12,708
Media (s)	144,325	54,456	46,701
FPS	12,430	32,944	

Tabla 11:Tiempos medios de proceso con particionado HW/SW

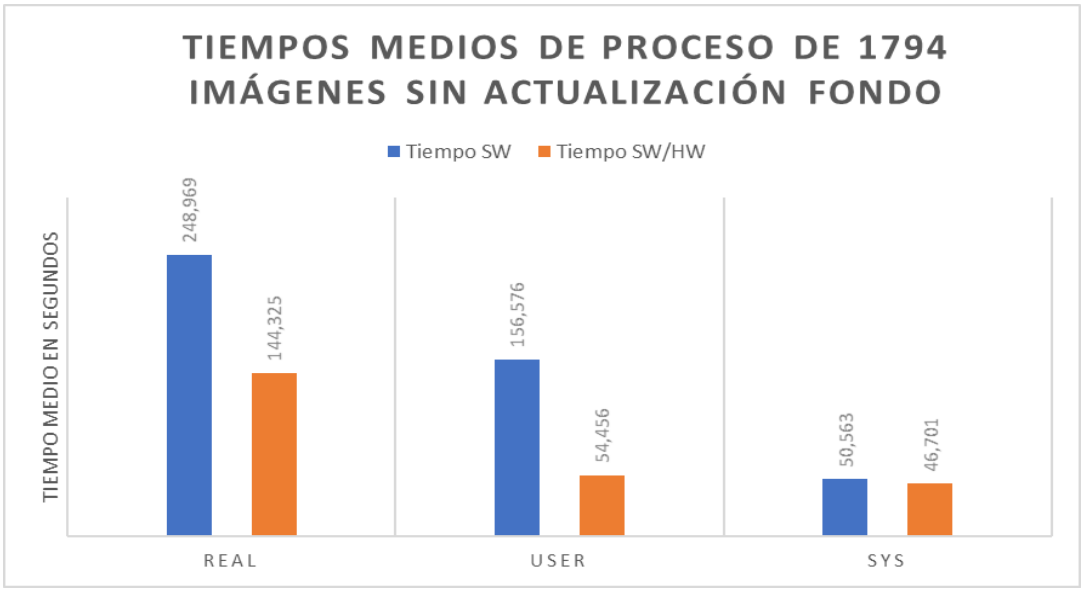


Figura 27: Gráfica comparativa tiempos medios de proceso una parte del algoritmo sin actualización de fondo

Con estos datos se ha conseguido un *speed-up*, para la medida *real*, de 1,725 y, para *user*, de 2,875. De esto se puede concluir que, fijándose en el tiempo total, la aceleración es ligeramente mayor (un 2,7%) a la esperada de la estimación ofrecida por SDSoC. Esto se debe a que el estimador cuenta con la sobrecarga que produce un SO pero no puede estimar perfectamente el tiempo que este tendrá esperando al programa por temas de planificación de tareas del propio sistema. Los FPS conseguidos no son muy altos debidos a que el SO operativo corriendo impide que se pueda conseguir la máxima eficiencia de los procesadores, haciendo que la parte que no está en HW no se procese todo lo rápido que se podría. Esto queda patente en la aceleración del tiempo que verdaderamente el programa se está ejecutando (columna de *user*), donde la aceleración supera a la estimada ya que aquí no se tiene en cuenta la sobrecarga del

sistema operativo. Además, para particionado HW/SW se consigue un FPS bastante buena, ya que superando el de una cámara normal. Cuantas más funciones se metiesen en HW, más aceleración se conseguiría, pero el impacto en el tiempo de proceso al introducir más sería cada vez menos apreciable. Debido a que el tiempo que consumen la CPU el resto de las funciones es cada vez menos significativo. Como se puede intuir la eliminación del SO también aportaría más velocidad al sistema. Aunque se deja como trabajo futuro.

El porcentaje de acierto para la primera parte ha sido del 98,16%, para la segunda del 95,48% y para la última del 97,71%. De lo que se extrae un porcentaje de acierto global medio del 97,12%. Este resultado es bastante bueno, ya que el porcentaje de error en el procesado de imágenes es menor del 5%. La principal fuente de error de decisión es debida a que, al estar desapareciendo o apareciendo un objeto, la clasificación manual observando cada imagen ha determinado que la última imagen con objeto era una y el algoritmo implementado consideraba una cercana pero no la misma. Otra posible causa de errores es cambios de iluminación en la imagen muy significativos y bruscos que hacen que el error de reconstrucción se dispare considerablemente en un corto periodo de tiempo, como sucede en la segunda tanda de imágenes. Por eso en esta parte las decisiones erróneas son más numerosas. Este efecto se ha intentado reducir restando distintos valores al valor de porcentaje de área detectada en cada frame. Para ello, se ha visto el error que introducía estos cambios bruscos y temporales. La forma adoptada para intentar disminuir su efecto ha sido restar un valor constante al porcentaje de área con objeto. Este valor se ha determinado viendo el porcentaje de área detectada cuando no hay objetos en la imagen en esta situación. De esta forma, se volvería a poder detectar correctamente la presencia, o no presencia, de objetos. Esta solución es muy poco flexible ya que solo sirve para este set de imágenes. Pero con un umbral dinámico se podría solucionar de forma sencilla. También se propondrá como trabajo futuro.

Tras realizar y analizar las medidas del algoritmo sin ninguna mejora se decidió añadir la posibilidad de que el fondo se fuese actualizando de forma dinámica con imágenes sin objeto. Se determinó que de cada 500 imágenes se valoraría una para ver si era apta para incluirla en el fondo y así actualizarlo. En este caso, los resultados obtenidos son los mostrados en las siguientes tablas.

Parte online SW 1794 imágenes	real	user	sys
Medida 1 (s)	242,482	157,199	28,727
Medida 2 (s)	273,580	159,373	52,859
Medida 3 (s)	317,305	159,420	96,175
Media (s)	277,789	158,664	59,253
FPS	6,458	11,307	

Tabla 12: Tiempos de ejecución partes con actualización de fondo solo SW

Parte online SW/HW 1794 imágenes	real	user	sys
Medida 1 (s)	141,599	54,569	29,237
Medida 2 (s)	166,236	54,777	51,831
Medida 3 (s)	76,512	54,403	19,571
Media (s)	128,116	54,583	33,546
FPS	14,003	32,867	

Tabla 13:Tiempos de ejecución partes con actualización de fondo particionado HW/SW

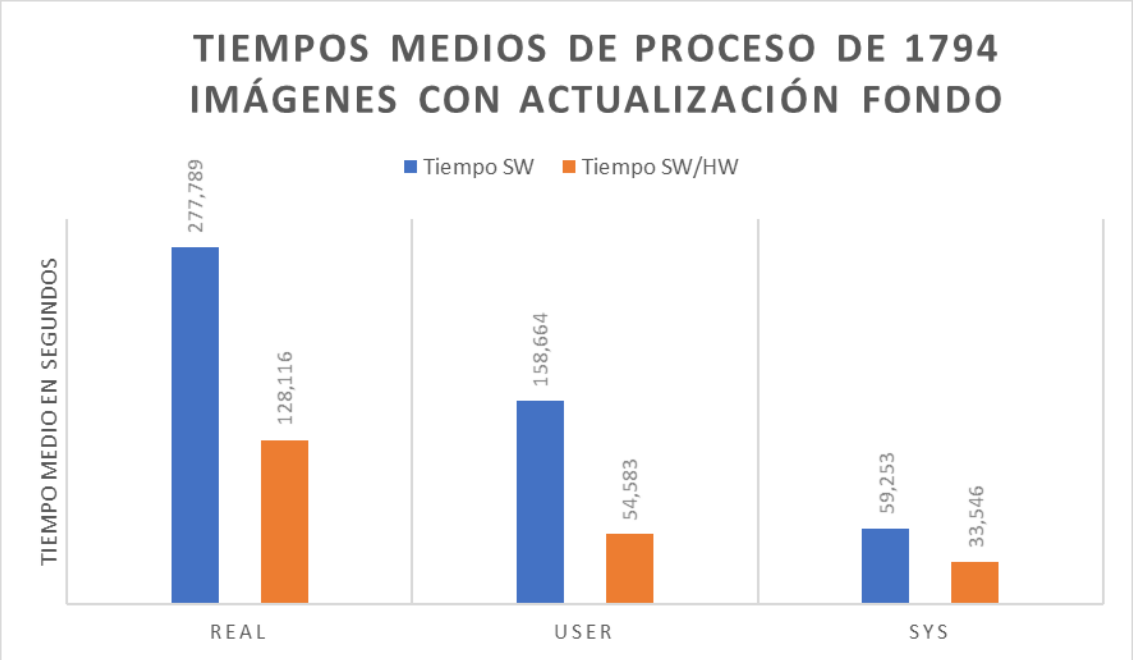


Figura 28: Gráfica comparativa tiempos medios de proceso una parte del algoritmo con actualización de fondo

El *speed-up* obtenido al introducir la actualización del fondo es de 2,168, para el caso de ver tiempos de *real*, y de 2,907 para el caso de *user*. Como se puede observar, la aceleración en el caso de tiempo *real* consumido es algo mayor, un 0,443. Esto se debe a dos factores principalmente. La primera es que el reparto de tiempos de CPU que el sistema operativo da al proceso en este caso no han sido los mismos. Ya que en la versión solo SW tarda más en ejecutarse que sin la mejora, pero en el caso del particionado tarda, de media, menos que sin la mejora. Esto se puede deber a que, al tener mayor carga computacional esta mejora del algoritmo, el SO le asigne mayor tiempo para que realice sus instrucciones al programa y así la demora no crezca mucho. En el caso de todo SW, el tiempo aumenta porque la carga computacional es mayor y aunque se le asigne más tiempo para que el programa pueda realizar su funcionalidad, el algoritmo no puede reducir tiempos. En cambio, en el particionado, le permite al HW no estar casi tiempo sin hacer nada porque al tener mayor tiempo para ejecutar SW, gracias a la planificación del SO, puede estar teniendo un uso más eficiente de los recursos HW. Lo que hace que el tiempo de ejecución sea menor.

Si se observa el tiempo de ejecución real, se observa que efectivamente hay una sobrecarga en el cómputo del algoritmo por esta actualización del fondo. En este caso el s*peed-up* mejora algo, ya que la actualización de fondo se ha paralelizado

completamente con procesos HW en el caso del particionado. Esto hace que el tiempo de proceso en este caso aumente en una medida muy pequeña (127 ms). En cambio en todo SW, la sobrecarga que introduce es más representativa, ya que suele actualizar dos veces el fondo aproximadamente en cada ejecución. Esto supone ejecutar dos partes offline más que sin esta mejora. Entonces, como el algoritmo acelerado no aumenta casi pero el sin acelerar si aumenta en mayor medida, el aceleración resultante es mayor a la obtenida sin mejora.

En este caso los porcentajes de acierto son: del 98,16% para la primera parte, del 92,59% para la segunda parte y del 88,96% para la tercera. Para la segunda parte, al hacerla con particionado salen 3 errores menos (92,75% de tasa resultante) y 1 error menos en la tercera parte (89,02 % de tasa). Obteniendo una tasa de acierto media del 95,24% para solo SW y del 95,32% para el particionado. Esta tasa es peor que la anterior sin mejora porque no se han eliminado las correcciones que se introducían antes para que el algoritmo se ajustase lo máximo posible al set de imágenes dado. Ya que se quería observar la influencia de la actualización de fondo sin modificar nada más. Así todo cambio que se observará en el algoritmo se pudiese identificar con la introducción de la actualización de fondo unívocamente. Es verdad que la tasa de acierto se ha reducido, pero sigue siendo bastante buena. La diferencia de errores entre solo SW y HW/SW se debe a que como tenemos una paralelización en el particionado puede que se estén procesando otros frames con la información de fondo anterior mientras que en solo SW hasta que no está el fondo totalmente actualizado, no se sigue procesando frames. Esto hace que la actualización tenga efecto más tarde y la adaptación al set de imágenes realizada con anterioridad tenga más efecto beneficioso para el caso que la actualización de fondo llega más tarde. La segunda y la tercera parte es donde se producen cambios de la tasa de aciertos con respecto al algoritmo sin mejora ya que en estas dos partes hay actualización de fondo. En la primera no se produce ninguna. Los fallos aumentan más en la última parte debido a que en esta se actualiza el fondo pero, como hay una adaptación del umbral que afecta a todo esta parte del set, provoca que este ajuste genere más error que eliminarlo. Esta comparativa de la tasa de aciertos entre la ejecución con y sin mejora queda plasmada en la Figura 29.

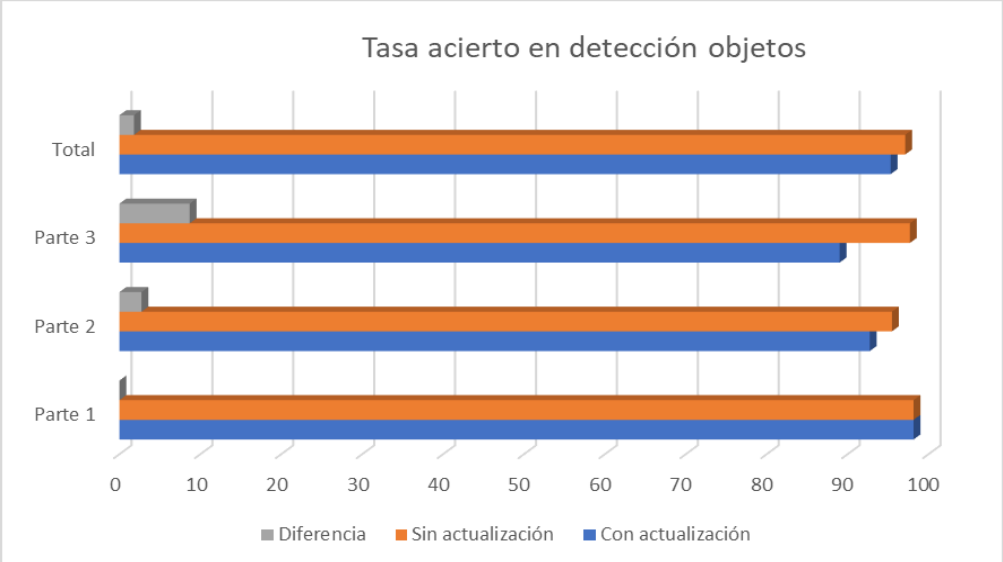


Figura 29: Comparativa tasa de acierto de algoritmo con y sin mejora

Para poder observar más claramente como afecta esta mejora a la aceleración del algoritmo y al FPS que es capaz de procesar imágenes, se incluyen las siguientes gráficas con las que se ilustrará lo comentado en este análisis de resultados. La diferencia es entre el valor que hay al incluir la mejora y el valor sin mejora. Se presentan los resultados para *real* y para *user* ya que son los que se han comentado previamente y tienen algún tipo de interés para el análisis.

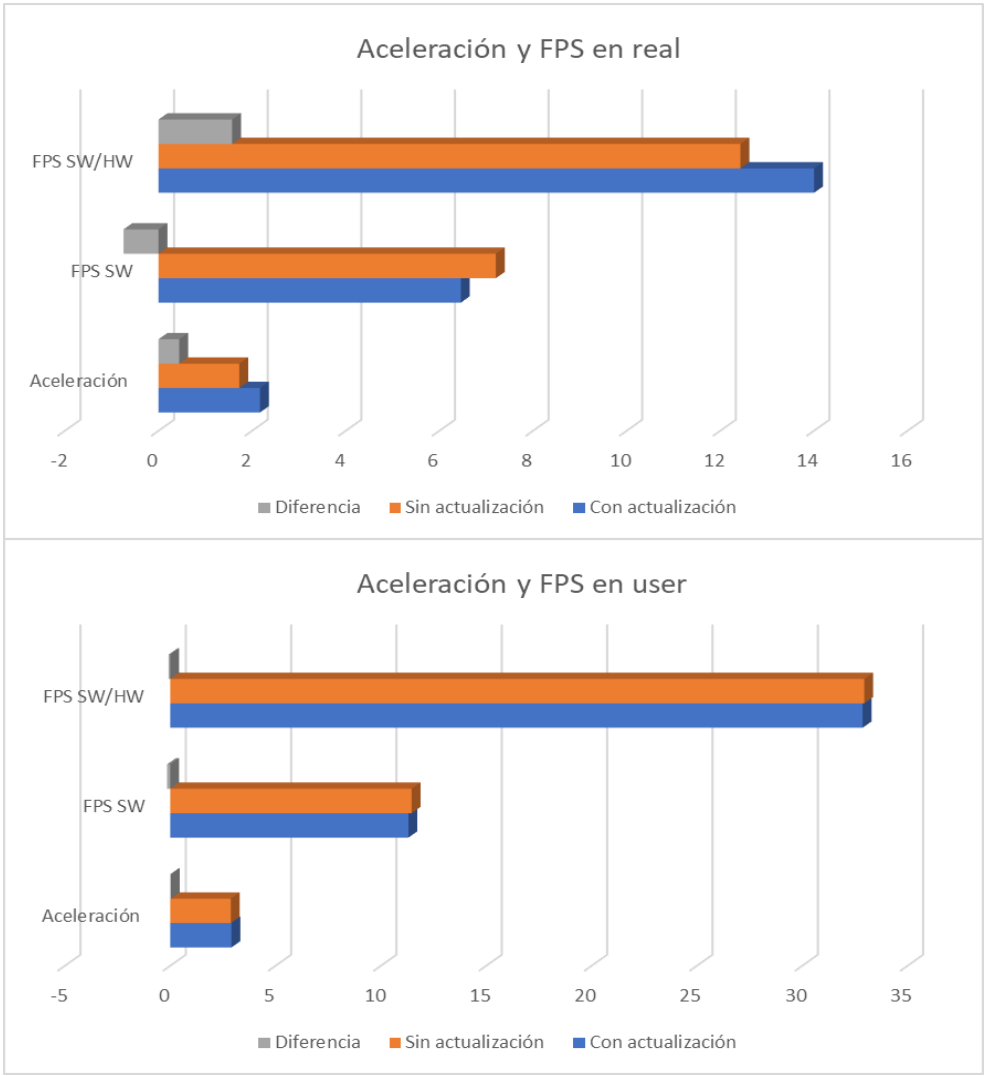


Figura 30: Aceleración y FPS antes y después de la mejora

7. Conclusiones y trabajos futuros

Al trabajar con la herramienta SDSoC para particionar un algoritmo en Hardware y Software, se han podido ver varias ventajas. La primera de ella es que, con tener conocimientos de un lenguaje de programación de alto nivel, como por ejemplo C++, y sabiendo unas nociones de las restricciones que presenta el HW a la hora de implementar algoritmos (no uso de memoria dinámica, bucles con finales definidos en compilación), es suficiente para poder realizar un diseño con particionado HW/SW. No es necesario conocimiento de lenguajes de descripción HW, como puede ser VHDL, o temas de sincronismo de los distintos bloques IP que forman el diseño de la parte HW. Tampoco hace falta saber cómo gestionar que haya operaciones en paralelo en HW y SW. Ya que de todo esto se encarga de organizarlo la propia herramienta de SDSoC.

Otra de ellas, sería que la elección de si tener una función en HW o SW es cuestión de seleccionar una opción y todo lo demás se encarga el programa de realizarlo automáticamente. Solo hay que indicarle, a través de pragmas, como serán las conexiones entre la parte lógica y la parte programable, si se considera oportuno para un mejor rendimiento o por no tener espacio en BRAM de la FPGA para copiar los datos ahí y trabajar con ellos. Lo que hace que el tiempo de desarrollo de un diseño se reduzca drásticamente. Porque no se tiene que crear el bloque IP manualmente desde cero y luego unir las distintas entradas salidas a la parte SW o a memoria compartida con sus correspondientes buses. También se pueden usar los pragma para guiar al programa en cómo debe implementar estas funciones en HW y así conseguir que la latencia de procesado sea aún menor. Además realiza, de forma autónoma, una paralelización de tareas HW/SW para que la aceleración del algoritmo sea mayor. Esto consigue que la aceleración final conseguida sea mayor que si al meter en HW funciones, la parte SW se quedará esperando a que el bloque HW acabase de realizar su operación sin hacer nada mientras.

También te permite estimar de forma rápida y sencilla cuántos recursos están consumidos de la parte HW en cada momento. Además del tiempo que tardará en procesarse cada una de las funciones que se están introduciendo en esta parte. Lo que permite al diseñador tener un control total del diseño y saber cuál es la aceleración que se está consiguiendo en cada momento. Además de saber el nivel de ocupación de FPGA para saber si puede o no introducir más funciones para acelerar.

En cuanto al algoritmo PCA, hay que remarcar que es bastante bueno para detectar objetos cuando se está en un entorno controlado: no hay cambios de luminosidad, la cámara esta estática y el fondo se mantiene constante. En cambio, si no se tiene actualización dinámica del fondo o del umbral de decisión, es muy sensible a cualquier pequeño cambio y la detección errónea de objeto se dispara. Si el cambio de luminosidad se produce de forma muy brusca, el fondo dinámico no tiene tiempo de adaptarse y habría un cierto periodo con error. Por lo que los cambios de luminosidad o fondo tienen que ser paulatinos para que se puedan asimilar sin producir mucho error. También hay que tener en cuenta que el algoritmo se ha adaptado en gran medida al set de imágenes que se han usado en el trabajo. Por lo que si se cambia las imágenes que se procesan el error puede aumentar considerablemente y habría que hacer una readaptación al nuevo set para

contener este error. Una forma de solucionarlo sería poniendo un umbral que se fuese adaptando a las imágenes que se tienen como entrada.

En cuanto a trabajos futuros se propone implementar el algoritmo sin hacer el uso de sistema operativo e introduciendo las imágenes en memoria a la hora de cargar el programa o pasarlas a través de comunicaciones remotas como puede ser Ethernet.

Trabajos futuros que se podrían hacer con respecto a las imágenes o al algoritmo PCA hay varios caminos a seguir. En cuanto a las imágenes, se podría proponer la modificación del algoritmo para procesar imágenes en color. De esta forma, resultaría más sencillo detectar los objetos completamente, ya que con solo el nivel de gris puede que un objeto quede camuflado con el fondo por tener un nivel de gris similar. Como pasa con el tronco de la persona que se pretende detectar en el set de imágenes usados para el trabajo. También se podría intentar incluir en el fondo objetos que se introducen en la escena después del arranque del sistema y van a permanecer estáticos por un periodo de tiempo suficientemente largo para considerarlo fondo mientras este ahí (por ejemplo una caja que se deja en la escena). También se podría hacer un umbral dinámico para que el algoritmo se adaptase a las condiciones de cada nuevo set de imágenes distintas y detecte que hay objetos nuevos en la imagen correctamente sin necesidad de hacer reajustes manuales, como lo de restar una variable al porcentaje de área detectada, en la variable que se usa para determinar si hay objeto en la imagen o no. Otro camino que se podría seguir sería el de introducir todas las funciones que se puedan meter en la parte HW hasta tener un uso de la FPGA cercano al 100%. Así se podrá tener la mayor parte del diseño en HW y conseguir la mayor aceleración posible. Esto permitiría ver como de rápido se puede procesar las imágenes en este algoritmo y ver si el esfuerzo extra que supone meter todo lo posible en HW compensa la mejora de aceleración conseguida con respecto a la actual. Ya que las siguientes funciones que se metan tendrán menor repercusión en la mejora global del algoritmo ya que representan una menor parte de su ejecución global.

8. Planos

Debido a que el código usado para implementar el algoritmo en C++ puede no ser del todo comprensible al leerlo, se ha optado por explicar la forma en la que cada función implementa su labor. Así se podrá comprender mejor el flujo del programa.

"PCA_offline.cpp"

El primer archivo que se va a explicar es "PCA_offline.cpp". En este, se encuentra la función *main*. Esta está dividida en dos partes: la parte Offline y la parte Online. En cuanto a la parte Offline, se ha seguido la secuencia de operaciones descrita en la presentación del algoritmo a realizar en este trabajo (Figura 11). Primero, se leen las 8 primeras imágenes del set, que se usarán para, a partir de ellas, calcular la matriz de transformación 'U'. La función *mat_to_array* permite ir guardando las imágenes que se van abriendo en un vector columna. Cada vector columna luego formará parte de una columna de la matriz I, que era la matriz que contenía todas las imágenes que se usarán como fondo.

Tras tener todas las imágenes de fondo guardadas en el array de entrada, se pasa a la función *parte_offline*, es la encargada de calcular la matriz U que se usará el resto del programa para pasar al espacio transformado. Se comienza calculando la media de cada uno de los píxeles con la función *mean_row*. Esta devolverá en 'mean' la media de cada una de las filas del array. Cada una de estas corresponde con un píxel de la imagen. Tras tener el valor medio de cada píxel, se resta a la matriz de entrada para tener una matriz con media cero, condición que imponía el algoritmo para los datos de entrada. A continuación, se transpone con una función creada a tal efecto. De esta forma, se pretende controlar el código que se usa para realizar cada una de las acciones del algoritmo. Ya que luego puede ser que se introduzcan en HW y así es más fácil introducir sentencias pragma donde se desee para conseguir la máxima optimización del rendimiento del sistema.

A continuación se obtiene la matriz *Input*, usada para calcular la matriz *V*, haciendo $A^t * A$. Esta se muestra por pantalla para poder ver si los valores que se van a introducir a SVD (la función que lo implementa es *dsvd*) se corresponden con los calculados en MATLAB y detectar posibles anomalías lo antes posible. Parte de la matriz U definitiva también se mostrará para ver que la transformación se va a hacer usando la misma matriz. Por lo que los resultados obtenidos al final del proceso deben ser los mismos si esto coincide. La función *ordena_matrix* ordena los autovalores de mayor a menor y, de forma paralela, va ordenando los autovectores por columnas correspondientes a cada autovalor. Una vez ordenada la matriz de autovectores, se realiza $U = A * V$ para obtener la matriz U completa, con todas las componentes principales. De esta se seleccionarán los primeros 'COMP'(número de componentes principales de interés) autovectores, que son los de mayor peso, para formar la matriz U final. Tanto para la matriz de autovectores de salida como para la matriz U final se tiene en cuenta correcciones de signo por columnas en el primer caso y por filas en el segundo, ya que a la hora de computar se observaron diferencias de signo entre la ejecución de MATLAB y la de C++ ante los mismos datos de entrada. Se optó por tomar como válidos y correctos los devueltos por MATLAB. Para concluir con la parte Offline, se muestra el

valor del RMSE obtenido al elegir 4 componentes como principales, valor justificado en la explicación del algoritmo.

Ahora se pasa a explicar la parte online. En ella, ya se hace un procesado de los datos de entrada. Con él se consigue determinar si hay objeto o no en la imagen y calcular un mapa de distancias para ver dónde está el objeto. Además se cuantifican las detecciones erróneas de objeto que se producen en la ejecución. Al mapa de distancias se le aplica un threshold para binarizarla y se le aplica una erosión y una dilatación para eliminar posibles ruidos. Después, se guarda en un archivo de imagen la imagen binarizada y en otro la que se le ha eliminado el ruido para poderlo visualizarlas por pantalla más adelante al conectar la SD al ordenador. Así se puede ver fácilmente los efectos beneficiosos que produce este procesado de la imagen.

Primero se declaran una serie de variables que van a ser necesarias para la ejecución de esta parte del programa. Tras ellas, se hace la transposición de la matriz U que se usará más adelante en la proyección de la imagen para reducir la dimensionalidad. El bucle acaba al llegar la variable 'pos', indica el número de frame que se está calculando en cada momento, al valor 5390. Ya que ese es el número de imágenes que hay en el set de imágenes que se está procesando en este caso. En el caso de que se decida particionar todo el set en otros más pequeños, como es en el caso de la toma de medidas realizada en este trabajo, el número cambia. Concretamente cada una de las partes acaba en 1802, 3596 y 5390 respectivamente. Para esta parte con describir la funcionalidad de una iteración del bucle es suficiente ya que son todas iguales.

Una vez que se entra en el bucle lo primero que se hace es leer la siguiente imagen a procesar con *cv::imread*. Después la trasformo de "cv::Mat" en un vector columna, de esta forma, se mantiene la misma forma que la de guardar cada imagen dentro de la matriz que contiene las imágenes de fondo (I) . A continuación, se resta la media calculada en la parte Offline y se proyecta haciendo $\Phi = U^t * \varphi$. Siendo Φ la imagen proyectada y φ la imagen de entrada a la que se le ha restado la media. Como los valores resultantes aquí son bastante elevados, se opta por normalizar los valores para que estén comprendidos entre -255 y 255. De esta forma se evita que al reconstruir los valores de salida se disparen mucho. En este punto, solo hay 4 valores ya que se ha considerado 4 como el valor de componentes principales en la matriz de transformación. Con esto, se puede observar la reducción de la dimensionalidad comentada como uno de los beneficios de este algoritmo. Tras normalizar la proyección, se vuelve reconstruir la imagen con $I_{recons} = U * I_{proyect}$. Esta reconstruida también se somete a una normalización para poder, tras sumarle la media restada anteriormente, comparar el error de reconstrucción con la imagen de entrada. La función *recons_error* devuelve el mapa de distancias de esta imagen. Con estos datos ya se puede obtener una imagen umbralizada a partir de un *threshold* dado y, más adelante, eliminar posibles ruidos que tenga con la erosión y dilatación. La función de umbralización se podría sustituir por la proporcionada por OpenCV pero como esa función no se puede ver cómo está programada, se prefiere construirla desde cero y así poder tener control absoluto sobre ella. A continuación se guardan en ficheros las imágenes umbralizadas y con el error eliminado. Tras ello, se calcula el porcentaje de área en la que se ha detectado objeto (blanco en la imagen binarizada) y se hacen ajustes para que el umbral funcione bien a la hora de detectar objeto. Al ya tener un porcentaje final bueno, se muestra por pantalla el porcentaje de

área que ocupa el objeto detectado en la imagen sin ruido. Dependiendo del valor de este, se decide si hay objeto o no en la imagen. Se compara esta decisión con la clasificación manual para ver si es buena o mala. En caso de ser errónea se contabiliza para el recuento de errores final. Se hace distinción entre si es falso positivo o falso negarivo para poder tener más información al final. Por último, se aumenta en una unidad el valor de la variable 'pos' y de 'img_proc'. Así se pordrá procesar la siguiente imagen en la siguiente iteración del bucle y ver cuántas imágenes se han procesado en una ejecución del programa, valor necesario para poder calcular tasas de acierto al final.

Una vez cada 500 frames se elige ese frame como posible candidato para formar parte del fondo. En caso de que el algoritmo decida que no hay objeto en ese frame, se procede a actualizar el fondo con ese. Para ello, se llama a la función *add_back*. Esta divide la imagen original de entrada en 4 partes y calcula la media de los píxeles de cada una de ellas y las compara con las medias de los píxeles de 'mean'. En caso de que en 3 o más partes de la imagen, la diferencia de medias sea mayor del 10% de la del fondo en esa parte, se devuelve un 1 para indicar que hay que actualizar el fondo ya que esta frame que se está analizando aportará información nueva al fondo. Esta información es que ha habido un cambio de luminosidad en la imagen. Si se decide entrar a actualizar el fondo, se suma a la matriz de fondo la media restada al principio del cálculo de la parte offline para volver a tener los valores originales de las imágenes. Después se incluye en la columna que más lleve sin actualizarse, como si fuese una FIFO este array, el frame bajo estudio. Una vez se tiene la nueva matriz 'A' se recalcula la parte offline para obtener la nueva matriz 'U' y se calcula su transpuesta. Se concluye incrementando en una unidad el valor 'col' para saber en la próxima entrada que imagen de fondo hay que reemplazar. Una vez calculado esto se vuelve al procesado de las imágenes que van llegando.

Tras acabar el bucle, hay que liberar la memoria reservada por los punteros al inicio de esta parte Online para que no se quede sin memoria el sistema al ejecutar varias veces el programa y no liberarla cada vez que se termina de usar. Esto se hace con la función *sds_free()*. Además se muestra por pantalla el número de falsos positivos y negativos que se han producido durante la ejecución y el número de errores totales. A partir del cual se calculará la tasa de acierto del algoritmo, que también se mostrará por pantalla. El código que realiza toda esta funcionalidad es el siguiente:

```
/*
 * PCA_offline.cpp
 *
 *  Created on: 31 may. 2019
 *      Author: daniel
 */


#include "PCA.hpp"
#include <iostream>
#include <unistd.h>
```

```
#include "common/xf_common.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgcodecs.hpp"
#include <opencv2/opencv.hpp>
typedef unsigned char uchar;




void mat_to_array(cv::Mat frame,short A[SIZE][MAX],int col)
{
      int i,j;

      for(i=0;i<ANCHO;i++)
            for(j=0;j<ALTO;j++)
                  A[i*ALTO+j][col]=frame.at<uchar>(j,i);
}
void mat_to_vector(cv::Mat frame,uchar A[SIZE][1])
{
      int i,j;
      for(i=0;i<ANCHO;i++)
            for(j=0;j<ALTO;j++)
                  A[i*ALTO+j][0]=frame.at<uchar>(j,i);
}

float is_object(cv::Mat in)
{
      float porcentaje;
      unsigned int i,blanco=0;
      uchar img[SIZE][1];
      cv::Mat out (ALTO,ANCHO,CV_8UC1);



      cv::erode(in,out,cv::Mat());
      cv::dilate(out,in,cv::Mat());
```

```
	mat_to_vector(in,img);
	for (i=0;i<SIZE;i++)
	{
		if(img[i][0]==255)
			blanco++;
	}
	porcentaje=(float)blanco/SIZE*100;
	return porcentaje;

}
char add_back(uchar new_img[SIZE][1], uchar med[SIZE][1])
{
	int A[4]={0,0,0,0};
	int Ab[4]={0,0,0,0};
	char dist=0;
	float meda,medb;
	int i=0,j=0;
	while(j<128)
	{
		for(i=0;i<128;i++)
		{
			A[0]=A[0]+new_img[i+j*256][0];
			Ab[0]=Ab[0]+med[i+j*256][0];
		}
		for(i=128;i<256;i++)
		{
			A[2]=A[2]+new_img[i+j*256][0];
			Ab[2]=Ab[2]+med[i+j*256][0];
		}
		j++;
	}
	while(j<256)
		{
			for(i=0;i<128;i++)
			{
				A[1]=A[1]+new_img[i+j*256][0];
				Ab[1]=Ab[1]+med[i+j*256][0];
```

```
                        }
                        for(i=128;i<256;i++)
                        {
                                A[3]=A[3]+new_img[i+j*256][0];
                                Ab[3]=Ab[3]+med[i+j*256][0];
                        }
                        j++;
                }
        for (i=0;i<4;i++)
        {
                meda=A[i]/(128*128);
                medb=Ab[i]/(128*128);
                if(abs(meda-medb)>(medb/10))
                        dist++;
        }
        if (dist>2)
                return 1;
        else
                return 0;
}
void parte_offline(short A[][MAX],uchar mean [][1], float U[][COMP])
{
        short At[MAX][SIZE];
        float Input[MAX][MAX];
        float eigval[MAX],suma=0,total=0;
        float V[MAX][MAX],Utemp[SIZE][MAX];
        int i,j;

                mean_row(A,mean);
                subtract_mean_row_off(A,mean);
                transposeA(A,At);
                mult_matrixIn(At,A,Input,MAX,SIZE,MAX);
                printf("\n\nInput= \n\t");
                for (j=0;j<MAX;j++)
                {
                        for(i=0;i<MAX;i++)
                        {
```

```
                printf("\t %8.0f \t",Input[j][i]);
        }
        printf("\n\t");
}
dsvd(Input,MAX,MAX,eigval,V);
ordena_matrix(eigval,V,MAX);
printf("\n\nAutovectores= ");
for (i=0;i<MAX;i++){
                printf("\n\t");
                for (j=0;j<MAX;j++)
                                {
                                   if(j==0||j==1||j==4||j==6)
                                          V[i][j]=-V[i][j];
                                   printf("\t %5.4f \t",V[i][j]);
                                }
}
printf("\n\nAutovalores= \n\t");
        for (i=0;i<MAX;i++){
                printf("\t %14.0f \n\t",eigval[i]);
                total=total+eigval[i];
                if(i<COMP)
                        suma=suma+eigval[i];

        }
        //transposeV(eigvect,V);
        mult_matrixU(A,V,Utemp,SIZE,MAX,MAX);
        for(i=0;i<SIZE;i++)
        {
                if(i==0||i==1||i==4)
                {
                        for(j=0;j<COMP;j++)
                        {
                                U[i][j]=-Utemp[i][j];
                        }
                }else
                {
                        for(j=0;j<COMP;j++)
```

```
                                        {
                                                U[i][j]=Utemp[i][j];
                                        }
                                }
                        }
                        printf("\n\nU= ");
                        for (i=0;i<20;i++){
                                printf("\n%d\t",i);
                                for (j=0;j<COMP;j++)
                                {
                                   printf("\t %5.4f \t",U[i][j]);
                                }
                        }
                        printf("\n\nEl   RMSE   es   %2.2f%%   cogiendo   %d
componentes.\n\n\n",suma/total*100,COMP);


        }


        int main(){
        /*********************************************************************
*****************************************************

        PARTE OFFLINE

        *********************************************************************
*****************************************************/
             short A[SIZE][MAX];
             float U[SIZE][COMP];
             uchar mean[SIZE][1],col;
             unsigned short pos=0;
             char file[40];
             char objeto[5390]= (inicialización omitida);


             cv::Mat frame;
             for(col=0;col<8;col++){
                    sprintf(file,"img19/image019_%05d.bmp",pos);
                    pos++;
                    frame=cv::imread(file,cv::IMREAD_GRAYSCALE);
                    mat_to_array(frame,A,col);
```

```
            }
            parte_offline(A,mean,U);


        /*************************************************************************
**************************************************

        PARTE ONLINE
        *************************************************************************
**************************************************/
            uchar img[SIZE][1],out[SIZE][1],MD[SIZE],img_dist[ALTO][ANCHO];
            short proyect_norm[COMP][1],recons_norm[SIZE][1];
            float Ut[COMP][SIZE],proyect[COMP][1];
            float porcentaje,object=1.15;
            short *imgmed=(short*)sds_alloc(SIZE*sizeof(short));
            float *recons=(float*)sds_alloc(SIZE*sizeof(float));
            unsigned short fpos=0,fneg=0,img_proc=0;

            uchar threshold=60;
            transposeU(U,Ut);

            while(pos<5390)
                  {

                        sprintf(file,"img19/image019_%05d.bmp",pos);
                        frame=cv::imread(file,cv::IMREAD_GRAYSCALE);
                        mat_to_vector(frame,img);
                        subtract_mean_row_on(img,mean,imgmed);
                        mult_matrix_onP(Ut,imgmed,proyect,COMP,SIZE,1);
                        normalize_P(proyect,COMP,proyect_norm);
                        mult_matrix_onR(  U  ,  proyect_norm  ,  recons,
SIZE,COMP,1);
                        normalize_R(recons,SIZE,recons_norm);
                        add_mean_row(recons_norm,mean,out);
                        recons_error(img,out,MD);
                        threshold_img(MD,img_dist,threshold);
                        cv::Mat disp (ALTO,ANCHO,CV_8UC1,img_dist);
                        sprintf(file,"out19/MD/MD_frame_%05d.bmp",pos);
```

```
                        imwrite(file,disp);
                        porcentaje=is_object(disp);
                        sprintf(file,"out19/proc/proc_frame_%05d.bmp",pos);
                        imwrite(file,disp);
                        if(pos>1878)
                              porcentaje=porcentaje-4;
                        if(pos<1978 && pos>1878)
                              porcentaje=porcentaje-30;
                        if (porcentaje<0)
                              porcentaje=0;
                        if (porcentaje>object)
                        {
                              printf("El porcentaje del area detectada del
frame %4d es: %2.2f%% (Hay objeto)\n", pos,porcentaje);
                              if (objeto[pos]!=1)
                                    fpos++;
                        }
                        else
                        {
                              printf("El porcentaje del area detectada del
frame %4d es: %2.2f%% (No hay objeto)\n", pos,porcentaje);
                              if (objeto[pos]!=0)
                                    fneg++;
                              if (pos%500==0)
                                    if(add_back(img,mean))
                                    {
                                          col&=7;
                                          add_mean_row_off(A,mean);
                                          mat_to_array(frame,A,col);
                                          parte_offline(A,mean,U);
                                          transposeU(U,Ut);
                                          col++;
                                    }
                        }

                        pos++;
                        img_proc++;
```

```
                }
            printf("El numero de falsos positivos es: %d \n",fpos);
            printf("El numero de falsos negativos es: %d \n",fneg);
            printf("El numero de errores totales es: %d \n",fpos+fneg);
            printf("La tasa de acierto es: %2.2f%% \n",(1-
((float)(fpos+fneg)/img_proc))*100);

            sds_free(imgmed);
            sds_free(recons);
            return 0;
        }
```

"multiplica_matrices.cpp"

El siguiente fichero por analizar es "multiplica_matrices.cpp". En este, se incluyen todas las funciones de multiplicación que se usan en el algoritmo. Las más interesantes son *mult_matrix_onP* y *mult_matrix_onR*. Ya que estás se implementarán en Hardware y tiene incluidas sentencias pragma. En el interior tienen sentencias de *PIPELINE* para hacer que el proceso de multiplicación se divida en varios pasos paralelos y así conseguir más velocidad de proceso. En el caso de *onP*, hay una y, en el de *onR*, dos. También hay que destacar el hecho de que en la cabecera de cada una hay dos pragma de SDSoC que me permiten decirle al compilador cómo es el acceso a cada variable de la función y la forma en la que se debe de implementar este acceso a los datos para que la información de salida se pueda usar fuera de esta función. Si no se diesen estas indicaciones, intentaría copiar todo en BRAM, cosa que no podrá hacer porque se quedará sin espacio. Ya que esta acción es la que elige por defecto SDSoC para trabajar con datos en funciones que se han movido a HW. El código de esta parte es:

```
#include "PCA.hpp"


void mult_matrixIn(short A[][SIZE] , short B[][MAX] , float C[MAX][MAX], int m
,int n, int c)
{
       int i,j,k;


       //OPERACION DE MULTIPLICACION
        for (i=0;i<m;i++)
              {
               for (j=0;j<c;j++)
                 { C[i][j]=0;
                      for (k=0;k<n;k++)
                       {
                             C[i][j]=C[i][j]+(float)A[i][k]*B[k][j];
                       }
```

```
                }

            }
}
void mult_matrixU(short A[][MAX] , float B[][MAX] , float C[][MAX], int m ,int
n, int c)
{
       int i,j,k;


       //OPERACION DE MULTIPLICACION

        for (i=0;i<m;i++)

            {
             for (j=0;j<c;j++)

               { C[i][j]=0;

                    for (k=0;k<n;k++)

                     {
                           C[i][j]=C[i][j]+A[i][k]*B[k][j];

                     }

               }

            }
}
#pragma SDS data access_pattern(A:SEQUENTIAL)
#pragma SDS data zero_copy(B[0:SIZE])

void mult_matrix_onP(float A[COMP][SIZE] , short B[SIZE] , float C[COMP][1],
int m ,int n, int c)
{
       int i,j,k;
       //OPERACION DE MULTIPLICACION

       loop_1: for (i=0;i<COMP;i++)

            {
             loop_2:for (j=0;j<1;j++)

               { C[i][j]=0;

                   loop_3: for (k=0;k<SIZE;k++)

                     {
                          #pragma HLS PIPELINE II=1 rewind

                           C[i][j]=C[i][j]+A[i][k]*B[k];

                     }

               }

            }
}
#pragma SDS data access_pattern(A:SEQUENTIAL)
#pragma SDS data access_pattern(C:SEQUENTIAL)
void mult_matrix_onR(float A[SIZE][COMP] , short B[COMP][1] , float C[SIZE],
int m ,int n, int c)
{
       int i,j,k;


       //OPERACION DE MULTIPLICACION
```

```
loop_1: for (i=0;i<SIZE;i++)
	{
	#pragma HLS PIPELINE II=1
	 loop_2: for (j=0;j<1;j++)

	   { C[i]=0;

		loop_3: for (k=0;k<COMP;k++)
		 {
		      #pragma HLS PIPELINE II=1
		       C[i]=C[i]+A[i][k]*B[k][j];

		 }
	   }

	}
 }
```

"norm_err.cpp"

En "norm_err.cpp", se encuentra el cuerpo de las funciones de normalización, medida de error de reconstrucción, obtención de matriz binaria a través del mapa de distancias y ordenar la matriz de salida de SVD de mayor a menor autovalor. La otra función que se van a implementar en Hardware (*normalize_R*) se encuentran en este archivo. Los pragmas usados en el interior también son los de *PIPELINE*. Aquí la de normalización tiene dos, uno por bucle. También tiene sentencias en la cabecera para indicar al compilador como tiene que hacer los buses de interconexión entre memoria y el bloque IP en la FPGA. La explicación de por qué se eligen estas sentencias se da en el inicio del apartado de optimización HW. La función *normalize_P* normaliza a un valor entre -255 y 255 la salida de la multiplicación usada para proyectar en el espacio transformado y con menor dimensionalidad. *Threshold_img* obtiene la imagen umbralizada a partir de un *threshold* pasado como argumento. Por último, *ordena_matrix* ordena la matriz de autovalores y va moviendo los autovectores asociados consecuentemente usando el método de la burbuja. El código perteneciente a esta parte es:

```
#include "PCA.hpp"


#pragma SDS data zero_copy(in[0:SIZE])
#pragma SDS data access_pattern(in:RANDOM, out:SEQUENTIAL)
void normalize_R(float in[SIZE],int nelem, short out[SIZE][1])
{

	float max=abs(in[0]),val;
	int i;

	loop_max:for(i=1;i<SIZE;i++)
	{
		#pragma HLS PIPELINE II=1
		val=abs(in[i]);
		cond:if (max<val)
			max=val;
	}
	loop_norm:for(i=0;i<SIZE;i++)
	{
		#pragma HLS PIPELINE II=1
		out[i][0]=in[i]/max*255;
	}
}
void normalize_P(float in[COMP][1],int nelem,short out[COMP][1])
```

```
{

	float max=abs(in[0][0]);
	int i;
	for(i=1;i<COMP;i++)
	{
		if (max<abs(in[i][0]))
			max=abs(in[i][0]);
	}
	for(i=0;i<COMP;i++)
		out[i][0]=in[i][0]/max*255;

}

void recons_error(unsigned char img[SIZE][1],unsigned char
out[SIZE][1],unsigned char MD[SIZE])
{
	unsigned int i;
	unsigned char diff;

	loop:for(i=0;i<SIZE;i++)
	{

		diff=abs(img[i][0]-out[i][0]);
		MD[i]=diff;
	}

}
void threshold_img(unsigned char MD[SIZE],unsigned char
OUT[ALTO][ANCHO],unsigned char thres)
{
	short i,j;
	for (i=0;i<ANCHO;i++)
	{
		for(j=0;j<ALTO;j++)
		{
			if (MD[i*ALTO+j]<thres)
				OUT[j][i]=0;
			else
				OUT[j][i]=255;



		}
	}

}
void ordena_matrix(float val[],float vect[][MAX],int dim)
{
	float temp;
	unsigned char mod=1,i,j;
	while (mod==1)
	{
		mod=0;
		for (i=0;i<(dim-1);i++)
		{
			if(val[i]<val[i+1] )
			{
				mod=1;
				temp=val[i];
				val[i]=val[i+1];
				val[i+1]=temp;
				for(j=0;j<dim;j++)
				{
					temp=vect[j][i];
					vect[j][i]=vect[j][i+1];
					vect[j][i+1]=temp;
				}
```

```
                }
            }
        }
         }
```

"transpose_mean.cpp"

El último fichero de código fuente que se ha escrito para la aplicación, ya que "svd.cpp" se ha obtenido de [Buja,1997], es "transpose_mean.cpp". Todas las funciones que contiene este se han seleccionado para procesar en SW. Hay dos partes diferenciadas: las funciones de transponer matrices y las del tratamiento de la media de cada fila. En este segundo bloque, hay cuatro funciones. Estas son: cálculo de la media (*mean_row*), restar la media en la parte offline a las 8 columnas de la matriz de las imágenes de fondo (*subtract_mean_row_off)*, resta de la media a la imagen que se ha tomado para procesar en la parte online (*subtract_mean_row_on*) y añadir la media a la imagen reconstruida para, en *recons_error*, compararla con la original (*add_mean_row*). Todas estas están implementadas con uno o dos bucles para recorrer todas las posiciones del array y realizar las operaciones necesarias, sumas o restas principalmente. En *mean_row*, se hace una división para hallar la media de la fila. El código de esta parte es:

```
#include "PCA.hpp"

void transposeA(short IN[][MAX],short OUT [][SIZE])
{
       int i,j;
       for(i=0;i<SIZE;i++)
       {
              for(j=0;j<MAX;j++)
              {
                     OUT[j][i]=IN[i][j];
              }
       }
}

void transposeU(float IN[][COMP],float OUT [][SIZE])
{
       int i,j;
       for(i=0;i<SIZE;i++)
       {
              for(j=0;j<COMP;j++)
              {
                     OUT[j][i]=IN[i][j];
              }
       }
}

void mean_row(short A[][MAX],unsigned char mean[][1])
{
        int i,j;
        short suma;
        for(i=0;i<SIZE;i++)
              {
               suma=0;
                     for(j=0;j<MAX;j++)
                     {
                            suma=suma+A[i][j];
                     }
              mean[i][0]=suma>>3;

              }
}
void subtract_mean_row_off(short Input[][MAX],unsigned char mean[][1])
{
```

```
        int i,j;

        for(i=0;i<SIZE;i++)
              {
              for(j=0;j<MAX;j++)
                     {
                            Input[i][j]=Input[i][j]-mean[i][0];
                     }
              }
}
void subtract_mean_row_on(unsigned char Input[SIZE][1],unsigned char
mean[SIZE][1],short out[SIZE])
{
        int i;

        for(i=0;i<SIZE;i++)
              {
                     out[i]=Input[i][0]-mean[i][0];
              }
}
void add_mean_row(short Input[][1],unsigned char mean[][1],unsigned char
out[][1])
{
       int i;
       for(i=0;i<SIZE;i++)
              {
                     out[i][0]=(unsigned char)(Input[i][0]+mean[i][0]);
              }
}
void add_mean_row_off(short Input[][MAX],unsigned char mean[][1])
{
       int i,j;
       for(i=0;i<SIZE;i++)
        {
              for(j=0;j<MAX;j++)
              {
                     Input[i][j]=Input[i][j]+mean[i][0];
              }
        }
         }
```

"PCA.hpp"

Por último hay un fichero de cabecera ("PCA.hpp"), en el que se incluyen todas los archivos de cabeceras comunes necesarios para la ejecución de todos los ficheros. Además se incluye todas las declaraciones de las funciones que se usan en el algoritmo y no están desarrolladas en "PCA_offline.cpp". También están las variables que se usan en todos los archivos comentados anteriormente para definir el tamaño de los arrays a usar y los límites de los bucles. Su valor se ha establecido con sentencias *#define*. Estas son:

- **MAX:** es la que define cuantas imágenes se cogen como fondo. Lo que implica el número máximo de dimensiones que habrá en el espacio transformado que se está creando.
- **ALTO:** define cuantos píxeles tiene de alto la imagen de entrada. Por lo que este número es el número de filas de la matriz que guarda la imagen.
- **ANCHO:** determina, en número de píxeles, el ancho de la imagen captada. Este corresponde con el número de columnas de la matriz que acumula la imagen.

- **SIZE:** es el número de píxeles que tiene la imagen. Esta será el tamaño del vector columna que guarde la imagen para ser procesada o acumulada como fondo.
- **DIM:** la dimensionalidad del espacio transformado, que coincide con el valor de *MAX*.
- **COMP:** establece el número de componentes que se van a considerar representativas en el programa. En este caso, el valor elegido es 4.

El código de la cabecera es:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "sds_lib.h"


#define MAX 8
#define ALTO 256
#define ANCHO 256
#define SIZE (ALTO*ANCHO)
#define DIM 8
#define COMP 4


void mult_matrixIn(short A[MAX][SIZE] , short B[SIZE][MAX] , float
C[MAX][MAX], int m ,int n, int c);
void mult_matrixU(short A[][MAX] , float B[][MAX] , float C[][MAX], int m
,int n, int c);
void mult_matrix_onP(float A[COMP][SIZE] , short B[SIZE] , float
C[COMP][1], int m ,int n, int c);
void mult_matrix_onR(float A[SIZE][COMP] , short B[COMP][1] , float
C[SIZE], int m ,int n, int c);
void transposeA(short IN[][MAX],short OUT [][SIZE]);
void transposeU(float IN[][COMP],float OUT [][SIZE]);
void mean_row(short A[][MAX],unsigned char mean[][1]);
void subtract_mean_row_off(short Input[][MAX],unsigned char mean[][1]);
void subtract_mean_row_on(unsigned char Input[SIZE][1],unsigned char
mean[SIZE][1],short out[SIZE]);
void add_mean_row(short Input[][1],unsigned char mean[][1],unsigned char
out[][1]);
void add_mean_row_off(short Input[][MAX],unsigned char mean[][1]);
void normalize_R(float in[SIZE],int nelem, short out[SIZE][1]);
void normalize_P(float in[COMP][1],int nelem,short out[COMP][1]);
void recons_error(unsigned char img[SIZE][1],unsigned char
out[SIZE][1],unsigned char MD[SIZE]);
void threshold_img(unsigned char MD[SIZE],unsigned char
OUT[ALTO][ANCHO],unsigned char thres);
int dsvd(float a[][MAX], int m, int n, float w[], float v[][MAX]);
void ordena_matrix(float val[],float vect[][MAX],int dim);
```

9. Presupuesto

Los costes de realización de este trabajo se van a dividir en el coste de los materiales y coste por mano de obra. En cuanto a los materiales a usar se divide entre licencias SW y los dispositivos necesarios para la realización del trabajo. Además se incluirá el coste de los materiales de oficina usados para este trabajo.

Coste material Recursos Hardware				
Equipo	**Precio**	**Duración**	**Uso**	**Total**
Ordenador con Intel Core i7	600€	5 años	9 meses	90€
Zedboard	450€	5 años	9 meses	67,5€
Cables conexión	10€	3 años	9 meses	2,5€
Subtotal recursos Hardware				**160€**

Coste material Recursos Software				
Programa	**Precio**	**Duración**	**Uso**	**Total**
MATLAB 2018.3	1.200€	4 años	9 meses	225€
SDSoC 2018.3	1.000€	5 años	9 meses	150€
Office 2016	270€	3 años	2 meses	15€
Windows 10	140€	3 años	9 meses	26,1€
Subtotal recursos Software				**416,1€**

Material	**Total**
Papel	30€
Encuadernación	70€
Tóner	200€
Total material	**300€**

Por lo que el total de coste de material asciende a:

Total recursos hardware	160€
Total recursos software	416,1€
Total material de oficina	300€
TOTAL MATERIALES	**876,1€**

El coste total de los materiales usados asciende a **ochocientos setenta y seis con diez céntimos.**

Los costes de mano de obra se calculan a partir de hora trabajada. Se incluyen los costes de Ingeniería, redacción y realización del libro.

Trabajo	Coste por horas	Total horas	Total
Diseño Ingeniería y redacción proyecto	35€	650 horas	22.750€
Realización del libro	15€	120 horas	1.800€
TOTAL MANO DE OBRA			**24.550€**

El coste total de la mano de obra asciende a **veinticuatro mil quinientos cincuenta euros.**

Con esto, ya se puede obtener el coste total de los materiales.

Coste de materiales	876,1€
Coste de mano de obra	24.550€
Coste total de ejecución material	**25.426,1€**

En el presupuesto de ejecución por contrata se incluirá tanto el beneficio por la ejecución del proyecto como los gastos ocasionados por la realización del mismo. Este se ha estimado en un recargo del 20% sobre el coste total de ejecución material.

Coste total de ejecución material	25.426,1€
20% de recargo	5.085,22€
Presupuesto de ejecución por contrata	**30.511,32€**

A este presupuesto hay que añadirle los honorarios por la realización del proyecto. Para concluir se añade el IVA, del 21%, y ya se obtendría el presupuesto final del proyecto.

Presupuesto de ejecución por contrata	30.511,32€
Honorarios realización proyecto (7%)	2.135,8€€
Presupuesto final	32.647,12€
21% de IVA	6.855,9€
PRESUPUESTO FINAL	**39.503,02€**

El importe final del proyecto asciende a **TREINTA Y NUEVE MIL QUINIENTOS TRES CON DOS CÉNTIMOS.**

10. Pliego de condiciones

Para la realización de este trabajo es necesario emplear los medios que se describirán en este apartado. Se hará una distinción entre los elementos HW necesarios y los SW.

En cuanto a los medios HW, se necesitan dos dispositivos: un PC y la placa de desarrollo ZedBoard de Digilent. Con respecto al PC, se van a indicar una serie de características mínimas para que todo los programas necesarios puedan ejecutarse sin problemas. El procesador tendrá que ser un Intel Core i7 de 8ª generación o superior. Referido al tema de memoria cabe destacar que se necesitarán, al menos, 8 GB de memoria RAM para poder soportar corriendo la máquina virtual y los programas de Xilinx a la vez. Se necesitarán 500 GB de disco duro para asegurarse de que no habrá problemas de quedarse sin memoria en medio de la compilación del programa. Ya que al ejecutar este procedimiento el SDSoC guarda mucha información y se puede llegar a necesitar, de forma eventual, mucho espacio para poder almacenar todo lo que demande el programa. Con respecto a gráficos, con una tarjeta gráfica Intel UHD Graphics 620 es suficiente. También será necesarios un puerto USB para usarlo como puerto serie de comunicación con la salida del programa que se esta ejecutando en la ZedBoard. Además tendrá que disponer de una ranura para conectar tarjetas SD. En ella se conectará la tarjeta SD que viene con la ZedBoard para cargar tanto el SO a usar como los programas que se quieran ejecutar y las imágenes que se quieran introducir para que sean procesadas por el algoritmo implementado. También será necesario un cable USB a micro USB para poder conectar la placa de desarrollo y el PC. Se usará para llevar la comunicación por UART entre los dos dispositivos. Otro cable necesario es el de Ethernet para poder realizar correctamente la depuración del programa en las primeras fases de desarrollo del proyecto.

Con respecto a los medios SW, habrá que disponer de varios programas para poder realizar todo lo expuesto en este trabajo. El primero que se deberá tener es el SDSoC. Con una versión 2018.3 como mínimo. Junto a él habrá que instalar el Vivado, para que el SDSoC pueda invocarle y realizar la implementación HW de forma correcta, y el HLS, usado para poder hacer una optimización de forma manual de las funciones que se incluyen en HW a través de sentencias pragma. Para que todo esto sea compatible los tres programas deben tener la misma versión. Además se necesitarán otros programas para poder comparar los resultados devueltos por la ejecución en placa con el resultado correcto. También se usarán para desarrollar el SW y comprobar que está bien desarrollado antes de pasarlo al particionado HW/SW. Para ello, se hace uso de MATLAB, versión 2018b, para ver los resultados que deben salir en la ejecución con las funciones creadas para este trabajo. También se hace uso de una máquina virtual, concretamente de la VMware Workstation 14, para correr un Ubuntu 18.04 LTS. Dentro de este se usará Eclipse para poder desarrollar todo el código que, tras su comprobación de que funciona correctamente, se usará en SDSoC para particionar el algoritmo en HW/SW. La versión de Eclipse usada es la 2019-06.

Con todo esto queda definido todo lo necesario para poder realizar este trabajo sin problemas de falta de recursos o no poder replicar ninguna parte del proyecto por falta de algún software o que no sea compatible.

11. Bibliografía

[Avnet,2014] ZedBoard Hardware User's Guide versión 2.2 27 January 2014 página 3.

[Suriano, 2017] L.Suriano Analysis of a Heterogeneous Multi-Core, Multi-HW-Accelerator-Based System Designed Using PREESM and SDSoC. Consultado: 9/9/2019. http://oa.upm.es/51778/1/recosoc17.pdf

[Kalb, 2016] T.Kalb Enabling Dynamic and Partial Reconfiguration in Xilinx SDSoC Consultado: 9/9/2019. http://tulipp.eu/wp-content/uploads/2019/01/reconfig2016_kalb.pdf

[Torres,2017] S. Torres Martínez, Implementación en FPGA usando SDSoC de un filtro espacio-espectral de mapas de clasificación hiperespectrales para detección de tumores cerebrales. Consultado: 13/06/2019. http://oa.upm.es/52464/1/TFG_SERGIO_TORRES_MARTINEZ.pdf

[López,2017] A. López García-Arias, Algoritmo de compresión de baja latencia en FPGA usando SDSoC. Consultado: 13/06/2019. https://repositorio.uam.es/bitstream/handle/10486/679338/Lopez_GarciaArias_Angel_tfg.pdf

[Bravo,2007] I.Bravo Muñoz, Arquitectura basada en FPGAs para la detección de objetos en movimiento, utilizando visión computacional y técnicas PCA. Apartado 3.4.

[Xilinx, 2019-2] Manual de usuario UG1233 (v2018.3) January 24, 2019 Consultado: 11/09/2019 https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1233-xilinx-opencv-user-guide.pdf

[Buja,1997] Buja, Cook, Asimov, Hurley. C Code for Computing a Grand Tour. Consultado: 20/06/2019. https://dicook.public.iastate.edu/JSS/paper/code.html

[Xilinx, 2019] Especificaciones Zynq-7000 series ZedBoard. Consultado: 1/07/2019. https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf

[Xilinx] PetaLinux Tools. Consultado:11/09/2019. https://www.xilinx.com/content/xilinx/en/products/design-tools/embedded-software/petalinux-sdk.html

[FreeRTOS,2019] Página principal del sistema operativo FreeRTOS. Consultado el 13/09/2019. https://www.freertos.org/

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá